# II. LINGUAGENS LIVRES DE CONTEXTO E AUTÔMATOS DE PILHA

## 2.1. Linguagens Livres de Contexto

**1. Definição.** Uma gramática $G$ é *livre de contexto* se $v$ é um único não terminal para toda produção $v \rightarrow w$ em $P$. Uma linguagem $L$ sobre algum alfabeto terminal $\Sigma$ é livre de contexto se pode ser gerado por uma gramática livre de contexto.

Então a linguagem dos palíndromos, a linguagem dos parênteses casados e a linguagem construída de cadeias de números iguais de *a*'s e *b*'s são todas livres de contexto, porque em todas foi mostrada uma gramática livre de contexto.

**2. Exemplo.** A seguinte gramática livre de contexto gera todas as cadeias sobre o alfabeto terminal 0, 1 com um número igual de 0's e 1's.

$$\Sigma = \{0,1\}$$
$$V = \{S, A, B\}$$

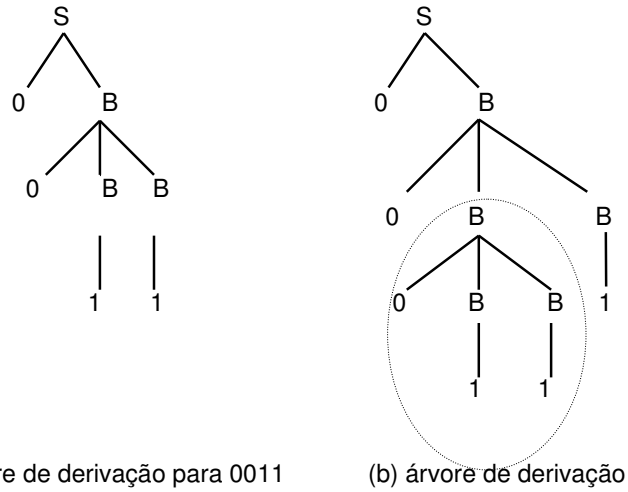
$P$ compreende as seguintes produções:

$$S \rightarrow 0B \mid 1A$$
$$A \rightarrow 0 \mid 0S \mid 1AA$$
$$B \rightarrow 1 \mid 1S \mid 0BB$$

As derivações livres de contexto têm uma representação em árvore muito útil e elegante. Por exemplo, a derivação das cadeias 0011 e 000111 usando a gramática do Exemplo 2 acima é dado na figura 1.

Se uma cadeia pode ser derivada legalmente por uma gramática *livre de contexto*, então podemos descrever esta derivação por uma árvore $T$ com as seguintes propriedades:

(1) A raiz é rotulada com o símbolo inicial $S$;
(2) Todo nó que não é uma folha é rotulado com uma variável - um símbolo de $V$;
(3) Todo nó que é uma folha é rotulado com um terminal - um símbolo de $\Sigma$ (ou possivelmente com $\lambda$);
(4) Se o nó $N$ é rotulado com um $A$, e $N$ tem $k$ descendentes diretos $N_1$, ..., $N_k$, rotulados com símbolos $A_1$, ..., $A_k$, respectivamente, então existe uma produção da gramática da forma $A \rightarrow A_1A_2...A_k$;
(5) Uma expressão derivada por alguma derivação pode ser obtida pela leitura das folhas da árvore associada com esta derivação, da esquerda para a direita.




(a) árvore de derivação para 0011 (b) árvore de derivação para 000111

**Figura 1**

Note que uma representação de árvore não ordena a aplicação de produções em uma derivação. Portanto, a Figura 1(a) representa uma das duas seguintes

$$S \Rightarrow 0B \Rightarrow 00BB \Rightarrow 001B \Rightarrow 0011$$
$$S \Rightarrow 0B \Rightarrow 00BB \Rightarrow 00B1 \Rightarrow 0011$$

A primeira derivação se refere à derivação mais a esquerda, já que o não terminal mais a esquerda em cada forma sentencial é sempre expandido primeiro. Analogamente, a segunda derivação é chamada de *derivação mais a direita*.

Dada uma árvore para uma derivação livre de contexto, define-se o *comprimento de um caminho* da raiz à folha como sendo o número de não terminais neste caminho. A *altura* de uma árvore é o comprimento de seu caminho mais longo. (Logo, na Figura 1(a), a altura da árvore é 3.)

Considere a sub-árvore assinalada na Figura 1(b). É uma árvore-$B$ legal; isto é, é uma árvore de derivação legal usando a gramática do Exemplo 2 exceto que a raiz é um $B$ e não um $S$. Agora se pegarmos qualquer árvore de derivação legal para uma cadeia nesta linguagem e substituirmos qualquer sub-árvore-$B$ nela com a sub-árvore assinalada, obtemos uma outra árvore de derivação legal. Este é o significado de "livre de contexto" do ponto de vista de representações de árvore. Vamos mostrar agora como a aplicação sistemática deste princípio de substituição de subárvores pode ser usado para estabelecer um resultado chave sobre a estrutura das linguagens livres de contexto.

Suponha que haja uma árvore de derivação $T$ para uma cadeia $z$ de terminais gerada por alguma gramática $G$, e suponha depois que o símbolo não terminal $A$ aparece duas vezes em algum caminho, como mostrado na Figura 2, onde $z = uvwxy$. Aqui a árvore-$A$ inferior deriva a cadeia terminal $w$, e a árvore-$A$ superior deriva a cadeia $vwx$. Como a gramática é livre de contexto, a substituição da árvore-$A$

superior pela árvore-*A* inferior não afeta a legalidade da derivação. A nova árvore deriva a cadeia *uwy*.

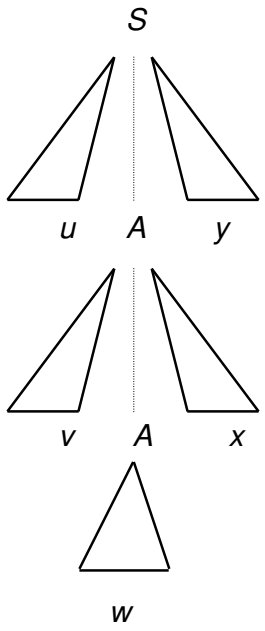



**Figura 2**. Uma árvore de derivação com o símbolo não terminal *A* aparecendo duas vezes no mesmo caminho.

Por outro lado, se substituirmos a árvore-*A* inferior pela árvore-*A* superior obtemos uma árvore de derivação legal para a cadeia *uvvwxxy*, que pode-se escrever como $uv^2wx^2y$. Esta substituição superior-para-inferior pode ser repetida qualquer número finito de vezes, obtendo-se o conjunto de cadeias $\{uv^nwx^ny \mid n \geq 0\}$. Toda LLC infinita deve conter infinitos subconjuntos de cadeias desta forma geral.

**3. Lema do Bombeamento para Linguagens Livres de Contexto.** (Também conhecido como Teorema *uvwxy*). *Seja L uma linguagem livre de contexto. Então existem constantes p e q dependentes de L apenas, tal que se z está em L com* $|z| > p$, *então z pode ser escrito como uvwxy de tal forma que*
(1) $|vwx| \leq q$;
(2) *no máximo um (v ou x) está vazio*; *e*
(3) *para todo* $i \geq 0$, *as cadeias* $uv^iwx^iy$ *estão em L*.

Note o seguinte:
(1) Dada uma linguagem gerada por uma gramática que não é livre de contexto, não se pode deduzir imediatamente que ela também não é gerada por uma gramática livre de contexto.
(2) Mas se uma linguagem infinita não obedece o lema do bombeamento para linguagens livres de contexto, ela não pode ser gerada por uma gramática livre de contexto.

## 2.2. Programas, Linguagens e *Parsing*

Como já foi visto, as linguagens livres de contexto são importantes para a ciência da computação porque elas representam um mecanismo razoavelmente adequado para especificar a sintaxe das linguagens de programação.

Seja o seguinte exemplo, as construções de programação **if-then** e **if-then-else** estão presentes em muitas linguagens de programação. Como uma primeira aproximação, considere a seguinte gramática:

$$S \to \textbf{if } C \textbf{ then } S \textbf{ else } S \mid \textbf{if } C \textbf{ then } S \mid a \mid b$$
$$C \to p \mid q$$

Aqui, $S$ é um (comando) não terminal, $C$ é um (condicional) não terminal, $a$ e $b$ são (comandos) terminais, $p$ e $q$ são (condições) terminais e **if**, **then** e **else** são (palavras reservadas) terminais.

Existem problemas com esta gramática. Ela gera a linguagem pretendida, mas de forma ambígua. Em particular,

**if** $p$ **then if** $q$ **then** $a$ **else** $b$

pode ser gerado de duas formas - ou como na Figura 3a ou 3b - correspondendo às duas interpretações diferentes do comando:

**if** $p$ **then** (**if** $q$ **then** $a$ **else** $b$)

e

**if** $p$ **then** (**if** $q$ **then** $a$) **else** $b$.

Do ponto de vista da programação, um jeito padrão de interpretar tais construções é associar cada comando **else** com o **if** mais próximo. Se uma gramática refletir esta preferência de programação acordada, então uma forma - neste caso a primeira - deve ser gerada pela gramática, e a alternativa deve ser excluída. Esta gramática é dada no próximo exemplo.

**1. Exemplo.** (Uma gramática alternativa **if-then-else**)

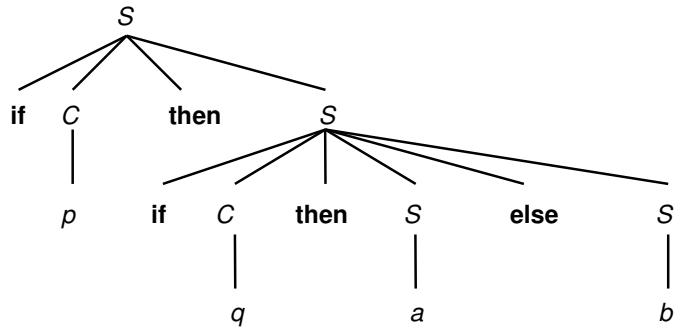
$$S \to S_1 \mid S_2$$
$$S_1 \to \textbf{if } C \textbf{ then } S_1 \textbf{ else } S_2 \mid T$$
$$S_2 \to \textbf{if } C \textbf{ then } S \mid \textbf{if } C \textbf{ then } S_1 \textbf{ else } S_2 \mid T$$
$$C \to p \mid q$$
$$T \to a \mid b$$

Esta gramática gera apenas uma derivação possível da cadeia

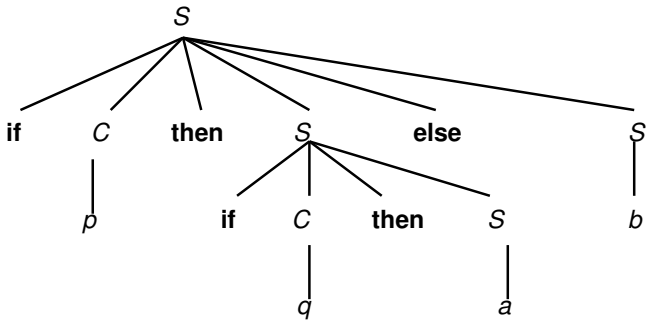
**if** $p$ **then if** $q$ **then** $a$ **else** $b$

que é a derivação na qual o comando **else** final, $b$, é ligado ao **if** mais próximo, como mostrado na Figura 4.

As gramáticas não ambíguas são essenciais para uma especificação sintática bem definida, de outra forma, um compilador poderia traduzir um programa de formas diferentes gerando resultados completamente diferentes. Logo, o estudo da ambigüidade é um aspecto prático e teórico importante da teoria das linguagens formais.




(a)




(b)

**Figura 3**. Duas árvores de derivação diferentes para **if** *p* **then if** *q* **then** *a* **else** *b*.

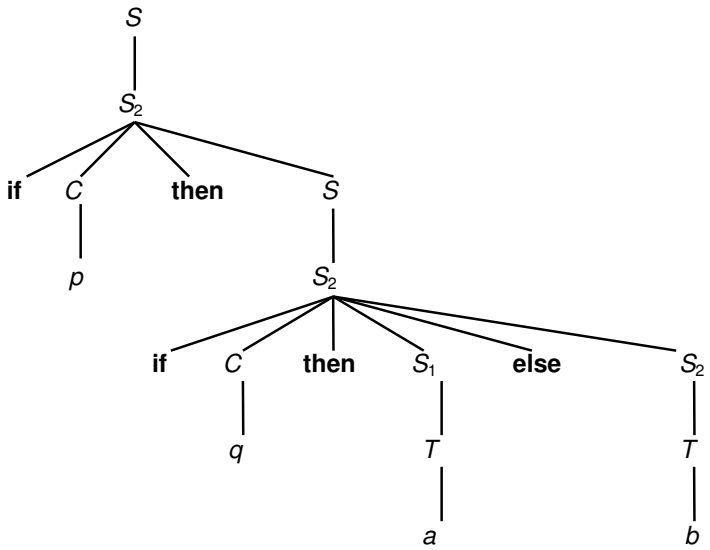



**Figura 4.** A única árvore de derivação de **if** *p* **then if** *q* **then** *a* **else** *b* usando a gramática do Exemplo 1.

**2. Definição.** Uma gramática livre de contexto $G$ é *ambígua* se alguma cadeia $x$ em $L(G)$ tem duas árvores de derivação distintas. Alternativamente, $G$ é ambígua se alguma cadeia em $L(G)$ tem duas derivações mais a esquerda (ou mais a direita) distintas. Uma gramática é *não ambígua* se ela não for ambígua e uma linguagem $L$ é *inerentemente ambígua* se *toda* gramática para $L$ for ambígua.

O exemplo do **if-then-else** ilustra como as gramáticas podem ser usadas para gerar construções de linguagens de programação. Descrições livres de contexto devem ter também uma segunda propriedade se elas forem um formalismo de linguagem de programação útil. Dada uma cadeia (um texto de programa) e uma linguagem (de programação) especificada por alguma gramática, deve ser possível construir uma árvore de derivação para a cadeia rapidamente e facilmente se a cadeia pertencer à linguagem, ou relatar um erro se a cadeia não for bem formada. O problema de fazer derivação de cadeia *backward* - dada uma cadeia, recuperar sua derivação - é conhecido como o *problema do parsing* para LLCs. Sua solução satisfatória é um dos pontos mais importantes no desenvolvimento da ciência da computação.

No momento, vamos nos preocupar com uma visão simplificada do *parsing*: dada uma cadeia $w$, como podemos dizer se $w$ é legalmente gerável por uma determinada GLC $G$? Existem duas estratégias básicas para resolver este problema. Uma estratégia, o *parsing* "bottom-up", considera a construção de uma árvore de *parse* para $x$, tomando por hipótese uma árvore de derivação para $x$ começando com o fundo (*bottom*) - as folhas da árvore - e trabalhando para cima na árvore até a raiz. A segunda estratégia básica, o *parsing* "top-down", trabalha de outra forma, tomando por hipótese o topo de uma árvore de derivação primeiro começando com a raiz. Vamos ilustrar estas duas técnicas com os próximos dois exemplos.

**3. Exemplo.** Fazer o *parsing* de expressões aritméticas de baixo para cima. Neste exemplo damos uma gramática para um fragmento da linguagem de expressões aritméticas e explicamos um algorítmo simples para fazer o *parsing* destas expressões "bottom-up". O símbolo inicial para a gramática é $E$.

$$
\begin{aligned}
&(1)\ E \to E + T\\
&(2)\ E \to T\\
&(3)\ T \to T * F\\
&(4)\ T \to F\\
&(5)\ F \to (E)\\
&(6)\ F \to 2
\end{aligned}
$$

Agora suponha que desejamos decidir se a cadeia

$$2 + 2 * 2$$

é gerada por esta gramática. Nossa abordagem é rodar a derivação *backward*, simulando uma derivação mais a direita como processamos a cadeia da esquerda para a direita. Os primeiros cinco passos na processo:

$$
\begin{aligned}
2 + 2 * 2 &\Leftarrow F + 2 * 2 \text{ (reverso 6)}\\
&\Leftarrow T + 2 * 2 \text{ (reverso 4)}\\
&\Leftarrow E + 2 * 2 \text{ (reverso 2)}\\
&\Leftarrow E + F * 2 \text{ (reverso 6)}\\
&\Leftarrow E + T * 2 \text{ (reverso 4)}
\end{aligned}
$$

E agora estamos num ponto crucial no *parsing*. Três caminhos são possíveis:

(1) converter $E + T$ em $E$, deixando $E * 2$;
(2) converter $T$ em $E$, deixando $E + E * 2$;
(3) converter 2 em $F$, deixando $E + T * F$.

As primeiras duas escolhas não são viáveis - elas levam a "dead ends", a partir das quais não há *parsing* bem sucedido possível. Escolha (3) que levará a um *parsing* bem sucedido, através de

(1) converter $T * F$ a $T$; e
(2) converter $E + T$ a $E$, o símbolo inicial da gramática.

Este exemplo ilustra o não determinismo possível no processo de *parsing*. O projeto de *parsers* eficientes - *parsers* que limitam, ou eliminam completamente, o tipo de não determinismo que vimos neste exemplo - é o maior componente da área da ciência da computação conhecida como análise sintática.

**4. Exemplo.** Este exemplo ilustra o segundo princípio geral de *parsing*: o *parsing* "top-down". Ilustramos este princípio mostrando como fazer o *parsing* de um fragmento de Pascal chamado de linguagem de programas-**while**. Esta linguagem simples permite comandos sucessor, predecessor e atribuição de zero, assim como loops while com testes simples da forma $x \neq y$. Um programa típico desta linguagem é o seguinte:

**begin** $y$ := 0;
**while** $x \neq y$ **do**
$y$ := **succ**($y$)
**end**

Este programa estabelece a valor de $y$ igual ao de $x$, isto é, o programa realiza o comando de atribuição $y := x$.

Vamos agora dar uma gramática simples para a sintaxe desta linguagem. Para esta gramática,

$\Sigma$ ={**begin**,**end**,**pred**,**succ**,:=,≠,**while**,**do**,;,(,),0,$x$, $y$}
$V = \{C, S, S_1, S_2, A, W, U, T\}$
Símbolo Inicial = $C$
Produções =
$C \rightarrow$ **begin** $S_1$ **end** {$C$ para comando composto}
$S_1 \rightarrow SS_2$
$S_2 \rightarrow ;S_1 \mid \lambda$
$S \rightarrow A \mid W \mid C$
$A \rightarrow U := T$ {$A$ para comando de atribuição}
$T \rightarrow$ **pred**($U$) | **succ**($U$) | 0
$W \rightarrow$ **while** $U \neq U$ **do** $S$ {$W$ para comando while}
$U \rightarrow x \mid y$

Simplificamos a linguagem dos programas-**while** para o propósito deste exemplo, assumindo que apenas duas variáveis *x* e *y* estão presentes na linguagem.

Vamos agora fazer o *parsing* do programa simples dado, de uma forma "top-down". Primeiro, como *C* é o símbolo inicial da gramática, todo *parse* legal deve começar

**begin** $S_1$ **end**

porque esta é a única produção *C*. Depois examinamos o texto do programa e verificamos que o **begin** deve permanecer. A estratégia é expandir a variável mais a esquerda. O próximo símbolo de entrada é *y*. Existe uma produção $S_1$, $S_1 \rightarrow S\ S_2$, e usando esta produção é possível alcançar o símbolo *y*:

**begin** $S\ S_2$ **end**

Agora vamos expandir *S*. Existem três regras *S*, mas apenas $S \rightarrow A$ é aplicável pois *A* pode alcançar uma variável enquanto que *W* leva a **while**..., e *C* leva a **begin**...:

**begin** $A\ S_2$ **end**

Existe apenas uma produção *A*:

**begin** $V$ := $T\ S_2$ **end**

*V* pode ser *x* ou *y*, e já que o próximo símbolo é *y*:

**begin** $y$ := $T\ S_2$ **end**

Agora temos casado com a entrada até :=, e temos que expandir o *T* para o próximo símbolo 0. Existem três produções *T* mas apenas $T \rightarrow 0$ é relevante:

**begin** $y$ := $0\ S_2$ **end**

O novo símbolo da entrada corrente é ;, e isto casa com a produção $S_2 \rightarrow ;S_1$, e falha com $S_2 \rightarrow \lambda$:

**begin** $y$ := 0; $S_1$ **end**

Como antes, $S_1 \rightarrow S\ S_2$ e neste caso, $S \rightarrow W$ é a única produção relevante:

**begin** $y$ := 0; $W\ S_2$ **end**

*W* gera o comando **while**:

**begin** $y$ := 0; **while** $V \neq V$ **do** $S\ S_2$ **end**

Os dois não terminais *V*'s casam com *x* e *y* respectivamente, a partir do texto:

**begin** $y$ := 0; **while** $x \neq y$ **do** $S\ S_2$ **end**

O próximo não terminal é *S*, e ele deve ser substituído por *A*, pois o próximo item é uma variável:

**begin** $y$ := 0; **while** $x \neq y$ **do** $A\ S_2$ **end**

Podemos substituir *A* pela expressão apropriada:

**begin** $y$ := 0; **while** $x \neq y$ **do** $y$ := **succ**($y$) $S_2$ **end**

Finalmente, convertemos $S_2$ na cadeia vazia, $\lambda$:

**begin** $y$ := 0; **while** $x \neq y$ **do** $y$ := **succ**($y$) **end**

Note que este processo é completamente determinístico: nenhum *backtracking* é necessário, e em todo estágio podemos predizer sem dificuldade qual produção é apropriada, dado o símbolo do texto corrente. A técnica "top-down" é chamada de *parsing LL*.

## 2.3. Gramáticas Livres de Contexto e a Língua Natural

No início desta parte foi dito que a teoria moderna das gramáticas formais segue em parte das teorias da linguagem natural que foram desenvolvidas pelo lingüis-

ta Noam Chomsky. Nesta seção, vamos descrever brevemente como as gramáticas formais podem ser usadas para descrever linguagem natural.

Suponha que identificamos as seguintes categorias da linguagem, as quais foram associadas com símbolos não terminais de uma gramática:

*S* – Sentença
*SN* – Sintagma Nominal
*SV* – Sintagma Verbal
*Adj* – Adjetivo
*Det* – Determinante
*V* – Verbo
*N* – Substantivo
*Prep* – Preposição
*SA* – Sintagma Adjetival
*SP* – Sintagma Preposicional

Dados estes símbolos, podemos escrever uma gramática livre de contexto que descreve a estrutura sintática de um pequeno subconjunto do Português.

$S \rightarrow SN\ SV$
$SN \rightarrow Det\ N \mid Det\ SA\ N \mid Det\ N\ SA \mid Det\ SA\ N\ SA$
$SA \rightarrow Adj\ SA \mid Adj$
$SV \rightarrow V_{lig}\ SP$
$SP \rightarrow Prep\ SN$

É claro que as palavras do Português constituem os símbolos terminais desta gramática, e portanto devemos adicionar produções da forma

$V_{lig} \rightarrow é \mid está...$
$Det \rightarrow o \mid a \mid os \mid as \mid um...$
$Prep \rightarrow em \mid sobre \mid para...$

etc. Dada esta gramática podemos derivar sentenças como “A grande bola vermelha está sobre a mesa” como na Figura 5.

Um problema imediato com esta gramática é a concordância de caso. A gramática gera sentenças do tipo “As grande bola vermelhas estão sobre as mesa”. Deve-se portanto, incluir as concordâncias de gênero e número.

Vários problemas lingüísticos surgirão a partir desta gramática simples. Distorções *não* livres de contexto. Chomsky argumenta que uma linguagem natural não pode ser gerada por uma gramática livre de contexto. Há a necessidade de algo mais complexo.

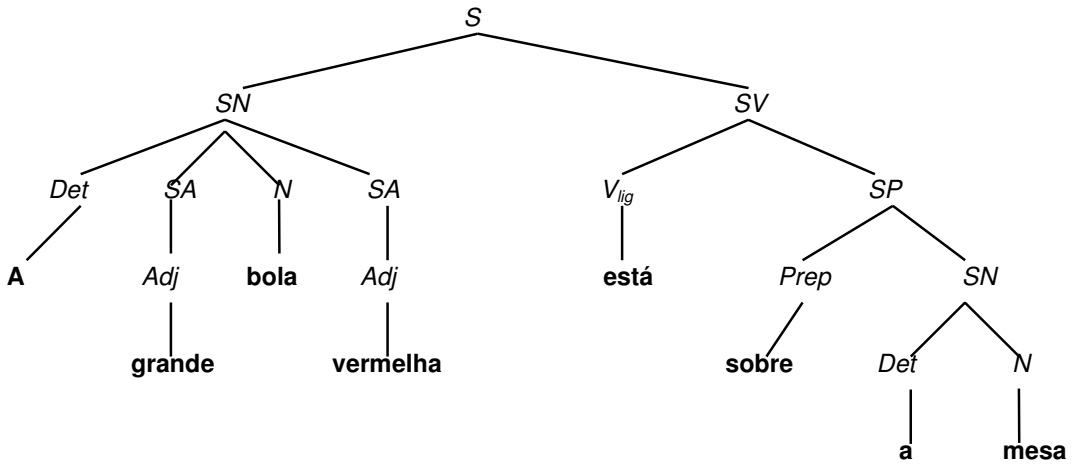



**Figura 5.** Árvore de derivação para "A grande bola vermelha está sobre a mesa".

## 2.4. Formas Normais para Gramáticas Livres de Contexto

Na matemática, uma *forma normal* é uma forma padrão de ver alguns objetos matemáticos. Ou seja, a fração ½ é uma forma normal para 5/10, 3/6 e 100/200.

Formas normais desempenham um papel importante na teoria da linguagem, e nesta seção discutiremos muitos resultados importantes de forma normal para gramáticas livres de contexto.

**1. Definição.** Dizemos que uma gramática livre de contexto $G = (V, \Sigma, S, P)$ está na *forma normal de Chomsky* (FNC) se cada produção estiver em uma das seguintes formas:

(i) $S \to \lambda$,
(ii) $A \to BC$, onde $A, B, C \in V$,
(iii) $A \to a$, onde $A \in V$ e $a \in \Sigma$.

Além disso, se $S \to \lambda$ estiver em $P$, então $B, C \in V - \{S\}$ na cláusula (ii).

**2. Teorema.** *Seja G uma gramática livre de contexto arbitrária. Então, existe uma gramática equivalente G' na forma normal de Chomsky*.

**3. Exemplo.** Dada a gramática

$$S \to ABC$$
$$C \to BaB \mid c$$
$$B \to b \mid bb$$
$$A \to a$$

Converte-se para a forma normal de Chomsky primeiro convertendo terminais do lado direito para variáveis e adicionando produções apropriadas:

$S \rightarrow ABC$
$C \rightarrow BA_1B \mid c$
$A_1 \rightarrow a$
$B \rightarrow b \mid B_1B_1$
$B_1 \rightarrow b$
$A \rightarrow a$

Finalmente separam-se os lados direitos com mais de duas variáveis:

$S \rightarrow AD$
$D \rightarrow BC$
$C \rightarrow BE \mid c$
$E \rightarrow A_1B$
$A_1 \rightarrow a$
$B \rightarrow b \mid B_1B_1$
$B_1 \rightarrow b$
$A \rightarrow a$

**4. Definição.** Dizemos que uma gramática livre de contexto está na *forma normal de Greibach* (FNG) se toda produção for da forma

$$A \rightarrow bW$$

onde $b \in \Sigma$ enquanto $W \in V^*$.

**5. Lema.** *Seja G uma gramática livre de contexto. Então existe uma gramática equivalente G', L(G) = L(G'), que não tem produções recursivas a esquerda, isto é, nenhuma produção da forma* $A \rightarrow Av$ *onde* $A \in V$ *e* $v \in (\Sigma \cup V)^*$ (Forma Normal sem Recursão à Esquerda).

As gramáticas livres de contexto (GLC) podem ter produções recursivas à esquerda, como em:

$A \rightarrow Av \mid w$

Estas produções derivam a cadeia $wv^*$, que também é derivada por:

$A \rightarrow wB \mid w$
$B \rightarrow vB \mid v$

*sem* recursões à esquerda.

(*) Para eliminar recursões à esquerda em geral, substitua

$A \rightarrow Av_1 \mid Av_2 \mid ... \mid Av_n \mid w_1 \mid ... \mid w_m$

que gera a expressão regular $(w_1 + ... + w_m)\ (v_1 + ... + v_n)^*$, por:

$A \rightarrow w_1 \mid ... \mid w_m \mid w_1B \mid ... \mid w_mB$
$B \rightarrow v_1 \mid ... \mid v_n \mid v_1B \mid ... \mid v_nB$

**6. Teorema.** *Seja G qualquer gramática livre de contexto. Então existe uma gramática equivalente G', isto é, L(G) = L(G'), na forma normal de Greibach*.

PROVA: Fixa-se a ordem das variáveis na gramática *G*: $V = \{ A_1, A_2, ..., A_n \}$. *G* é *crescente* se toda produção de *G* for da forma

(1) $A_j \rightarrow A_k v$, com $j < k$; ou
(2) $A_j \rightarrow av$, com $a \in \Sigma$.

Considerando que *G* tem produções ou da forma

$A \rightarrow A_{i1} ... A_{in}$, com $A_{i's} \in V$ ou
$A \rightarrow b$, com $b \in \Sigma$.

Se todas as produções de $A_1$ são *crescentes*, continua-se em $A_2$. De outra forma, há uma regra $A_1$ recursiva à esquerda que é substituída como acima (*). Daí vai-se para as produções $A_2$, substituindo por $A_1$ quando é encontrada uma produção da forma $A_2 \rightarrow A_1x$, e então eliminam-se recursões à esquerda em $A_2$, se houver. E assim por diante até $A_n$. Então faz-se a substituição de volta *de cima para baixo* pondo *G* na FNG. Finalmente, aplica-se a substituição de novo, às novas variáveis introduzidas quando a recursão à esquerda foi eliminada. ◊

*Exemplo*:

$A_1 \rightarrow A_2A_2 \mid 0$ → é *crescente*
$A_2 \rightarrow A_1A_2 \mid 1$ → não é *crescente*

Como as produções $A_1$ são *crescentes*, não se mexe nelas. Deve-se mexer nas produções $A_2$, substituindo $A_1$ pelas suas produções:

$A_1 \rightarrow A_2A_2 \mid 0$
$A_2 \rightarrow A_2A_2A_2 \mid 0A_2 \mid 1$

Observe que agora, a primeira produção $A_2$ tem recursão à esquerda. Seja $v = A_2A_2$ e $w = 0A_2$, já que $A \rightarrow Av \mid w$ deve ser transformado em $A \rightarrow wB \mid w$ e $B \rightarrow vB \mid v$, vem:

$A_1 \rightarrow A_2A_2 \mid 0$
$A_2 \rightarrow 0A_2 \mid 1 \mid 0A_2B \mid 1B$
$B \rightarrow A_2A_2 \mid A_2A_2B$

Agora tanto as produções $A_1$ quanto as produções $A_2$ são *crescentes*. Deve-se colocar as produções na FNG, fazendo aparecer um símbolo terminal à esquerda do lado direito de cada produção:

$A_1 \rightarrow 0A_2A_2 \mid 0A_2BA_2 \mid 1A_2 \mid 1BA_2 \mid 0$
$A_2 \rightarrow 0A_2 \mid 0A_2B \mid 1 \mid 1B$

E em B deve-se eliminar variáveis na posição mais à esquerda do lado direito:

$A_1 \rightarrow 0A_2A_2 \mid 0A_2BA_2 \mid 1A_2 \mid 1BA_2 \mid 0$
$A_2 \rightarrow 0A_2 \mid 0A_2B \mid 1 \mid 1B$
$B \rightarrow 0A_2A_2 \mid 0A_2BA_2 \mid 1A_2 \mid 1BA_2 \mid 0A_2A_2B \mid 0A_2BA_2B \mid 1A_2B \mid 1BA_2B$

**7. Exemplo.** Converta a seguinte gramática à Forma Normal de Greibach.

$S \rightarrow SSS \mid RS \mid 0$
$R \rightarrow RR \mid SR \mid 1$

*Resolução*:

1) Ordena-se o conjunto de variáveis: $V = \{S, R\}$
2) Todas as produções de S são crescentes. Eliminam-se as recursões à esquerda das produções de S:
   Como as produções $A \rightarrow Av \mid w$ derivam a cadeia $wv^*$, que também é derivada por:
   $A \rightarrow wB \mid w$
   $B \rightarrow vB \mid v$
   sem recursões à esquerda. Fazendo v = SS, $w_1$ = RS e $w_2$ = 0, tem-se:
   $S \rightarrow RSB \mid RS \mid 0B \mid 0$
   $B \rightarrow SSB \mid SS$
3) Substitui-se o S nas produções de R, pois a segunda produção de R ($R \rightarrow SR$) não é crescente:
   $R \rightarrow RR \mid RSBR \mid RSR \mid 0BR \mid 0R \mid 1$
4) Agora, todas as produções de R são crescentes. Eliminam-se as recursões à esquerda das produções de R. Fazendo $v_1$ = R, $v_2$ = SBR, $v_3$ = SR, $w_1$ = 0BR, $w_2$ = 0R e $w_3$ = 1, tem-se:
   $R \rightarrow 0BRC \mid 0BR \mid 0RC \mid 0R \mid 1C \mid 1$
   $C \rightarrow RC \mid R \mid SBRC \mid SBR \mid SRC \mid SR$
5) As produções atualizadas são:
   $S \rightarrow RSB \mid RS \mid 0B \mid 0$
   $B \rightarrow SSB \mid SS$
   $R \rightarrow 0BRC \mid 0BR \mid 0RC \mid 0R \mid 1C \mid 1$
   $C \rightarrow RC \mid R \mid SBRC \mid SBR \mid SRC \mid SR$
   Para gerar a FNG precisa-se eliminar o R nas duas primeiras produções de S, substituindo-os pelas produções de R:
   $S \rightarrow$ 0BRCSB | 0BRSB | 0RCSB | 0RSB | 1CSB | 1SB | 0BRCS | 0BRS | 0RCS | 0RS | 1CS | 1S | 0B | 0
   Agora todas as produções de S estão na FNG. Então, elimina-se o S à esquerda das produções de B, substituindo-os pelas produções de S:
   $B \rightarrow$ 0BRCSBSB | 0BRSBSB | 0RCSBSB | 0RSBSB | 1CSBSB | 1SBSB | 0BRCSSB | 0BRSSB | 0RCSSB | 0RSSB | 1CSSB | 1SSB | 0BSB | 0SB | 0BRCSBS | 0BRSBS | 0RCSBS | 0RSBS | 1CSBS | 1SBS | 0BRCSS | 0BRSS | 0RCSS | 0RSS | 1CSS | 1SS | 0BS | 0S
   Faz-se o mesmo pelas produções de C:
   $C \rightarrow$ 0BRCC | 0BRC | 0RCC | 0RC | 1CC | 1C | **0BRC** | 0BR | **0RC** | 0R | **1C** | 1 | 0BRCSBBRC | 0BRSBBRC | 0RCSBBRC | 0RSBBRCBRC | 1CSBBRC | 1SBBRC | 0BRCSBRC | 0BRSBRC | 0RCSBRC | 0RSBRC | 1CSBRC | 1SBRC | 0BBRC | **0BRC** | 0BRCSBBR | 0BRSBBR | 0RCSBBR | 0RSBBR |

1CSBBR | 1SBBR | 0BRCSBR | 0BRSBR | 0RCSBR | 0RSBR | 1CSBR | 1SBR | 0BBR | **0BR** | **0BRCSBRC** | **0BRSBRC** | **0RCSBRC** | **0RSBRC** | **1CSBRC** | **1SBRC** | 0BRCSRC | 0BRSRC | 0RCSRC | 0RSRC | 1CSRC | 1SRC | **0BRC** | **0RC** | **0BRCSBR** | **0BRSBR** | **0RCSBR** | **0RSBR** | **1CSBR** | **1SBR** | 0BRCSR | 0BRSR | 0RCSR | 0RSR | 1CSR | 1SR | **0BR** | **0R**

6) A gramática na Forma Normal de Greibach (eliminando as produções repetidas (em negrito)):

S $\rightarrow$ 0BRCSB | 0BRSB | 0RCSB | 0RSB | 1CSB | 1SB | 0BRCS | 0BRS | 0RCS | 0RS | 1CS | 1S | 0B | 0

B $\rightarrow$ 0BRCSBSB | 0BRSBSB | 0RCSBSB | 0RSBSB | 1CSBSB | 1SBSB | 0BRCSSB | 0BRSSB | 0RCSSB | 0RSSB | 1CSSB | 1SSB | 0BSB | 0SB | 0BRCSBS | 0BRSBS | 0RCSBS | 0RSBS | 1CSBS | 1SBS | 0BRCSS | 0BRSS | 0RCSS | 0RSS | 1CSS | 1SS | 0BS | 0S

R $\rightarrow$ 0BRC | 0BR | 0RC | 0R | 1C | 1

C $\rightarrow$ 0BRCC | 0BRC | 0RCC | 0RC | 1CC | 1C | 0BR | 0R | 1 | 0BRCSBBRC | 0BRSBBRC | 0RCSBBRC | 0RSBBRCBRC | 1CSBBRC | 1SBBRC | 0BRCSBRC | 0BRSBRC | 0RCSBRC | 0RSBRC | 1CSBRC | 1SBRC | 0BBRC | 0BRCSBBR | 0BRSBBR | 0RCSBBR | 0RSBBR | 1CSBBR | 1SBBR | 0BRCSBR | 0BRSBR | 0RCSBR | 0RSBR | 1CSBR | 1SBR | 0BBR | 0BRCSRC | 0BRSRC | 0RCSRC | 0RSRC | 1CSRC | 1SRC | 0BRCSR | 0BRSR | 0RCSR | 0RSR | 1CSR | 1SR

6 produções na GLC original - 95 produções na GLC na forma normal de Greibach.

## 2.5. Autômatos de pilha

Nesta seção vamos estabelecer uma das equivalências mais importantes na ciência da computação teórica: prova-se que uma linguagem é livre de contexto se e somente se algum autômato de pilha possa aceitar a linguagem de uma forma precisa. Este resultado tem importância prática e teórica, porque um armazém de pilha é a base para muitos algoritmos usados no *parsing* de linguagens livres de contexto.

Uma *pilha*, familiar em ciência de computação, é uma estrutura *last-in first-out* com uma operação "push" que adiciona à pilha e uma operação "pop" que remove o elemento do topo da pilha, se existir.

A noção "pura" de uma pilha, descrita informalmente acima, aumentada pela noção de transição de estado *não determinístico*, provê um modelo de autômato completo para linguagens livres de contexto. Entretanto, certa estrutura adicional é também conveniente, e os próximos três exemplos mostram porque isto é assim.

**1. Exemplo.** Considere a linguagem

$$\{ w \mid w \in (a + b)^*, w \text{ tem um número igual de } a\text{'s e } b\text{'s}\}.$$

Esta linguagem é informalmente aceitável por uma pilha usando o seguinte algoritmo. Inicialmente, a pilha está vazia. Percorra a cadeia *w* da esquerda para a direita, e realize as seguintes operações, baseado no símbolo corrente no topo da pilha:

(1) se a pilha estiver vazia e o símbolo corrente de *w* for um *a*, ponha *A* na pilha;
(2) se a pilha estiver vazia e o símbolo corrente de *w* for um *b*, ponha *B* na pilha;
(3) se o símbolo no topo da pilha for um *A* e o símbolo corrente de *w* for um *a*, coloque (*push*) um outro *A* na pilha;
(4) se o símbolo no topo da pilha for um *B* e o símbolo corrente de *w* for um *b*, coloque (*push*) um *B* na pilha;
(5) se o símbolo no topo da pilha for um *A* e o símbolo corrente de *w* for um *b*, retire (*pop*) da pilha;
(6) se o símbolo no topo da pilha for um *B* e o símbolo corrente de *w* for um *a*, retire (*pop*) da pilha;

Este algoritmo informal julgará corretamente a legalidade de cadeias desta linguagem da seguinte forma: uma cadeia *w* tem um número igual de *a*'s e *b*'s se e somente se, depois de processar *w*, a pilha estiver vazia.

Abaixo escrevemos um "programa" de pilha para o algoritmo informal descrito acima. Foi incluído nesta descrição um símbolo especial, $Z_0$, para denotar o fundo da pilha. Então, usou-se a notação $\langle\ x, D, v\ \rangle$ para significar "se *x* é o próximo símbolo da cadeia de entrada *w* e *D* é o símbolo no topo da pilha, então substitua *D* pela cadeia *v*." Adotamos a convenção de escrever a pilha como uma cadeia de tal forma que o topo da pilha se torne a esquerda da cadeia. Então a descrição informal precedente pode ser rescrita como se segue:

(1) $\langle\ a, Z_0, AZ_0\ \rangle$
(2) $\langle\ b, Z_0, BZ_0\ \rangle$
(3) $\langle\ a, A, AA\ \rangle$
(4) $\langle\ b, B, BB\ \rangle$
(5) $\langle\ a, B, \lambda\ \rangle$
(6) $\langle\ b, A, \lambda\ \rangle$
(7) $\langle\ \lambda, Z_0, \lambda\ \rangle$

Note que esta descrição segue da consideração informal dada anteriormente exceto que foi introduzido o símbolo $Z_0$ como o marcador do fundo da pilha. Devemos introduzir a "regra-λ" (7) que, quando não houver entrada, apaga o marcador de fundo $Z_0$. Dizemos que uma cadeia é aceita se depois da cadeia ter sido completamente percorrida, a pilha estiver vazia.

**2. Exemplo.** Considere a linguagem $\{wcw^R \mid w \in (a + b)^*\}$. Esta linguagem, como se verá, é reconhecível por um autômato do tipo do último exemplo, mas um jeito mais conveniente de reconhecê-la é aumentar a maquinaria da pilha com uma estrutura de estados finitos consistindo de dois estados, um chamado *push* para processar a primeira metade da cadeia, o outro, chamado *pop*, para a segunda metade. A entrada *c* engatilha a transição de *push* para *pop*. Vamos usar agora a notação $\langle\ q, x, D, q', v\ \rangle$ para significar "se a máquina de estados estiver no estado *q*, *x* for o próximo símbolo da cadeia de entrada e *D* for o símbolo no topo da pilha, então a máquina de estados pode mudar para o estado *q*' e substituir *D* pela cadeia *v*." Devemos usar

o símbolo $Z$ para um símbolo de pilha arbitrário. Então a instrução $\langle$ *push*, *a*, *Z*, *push*, *AZ* $\rangle$ é o resumo para as três instruções

$$\langle \textit{push}, a, Z_0, \textit{push}, AZ_0 \rangle$$
$$\langle \textit{push}, a, A, \textit{push}, AA \rangle$$
$$\langle \textit{push}, a, B, \textit{push}, AB \rangle$$

Usando esta notação aumentada, podemos descrever o autômato de pilha (APN) como:

$$\langle \textit{push}, a, Z, \textit{push}, AZ \rangle$$
$$\langle \textit{push}, b, Z, \textit{push}, BZ \rangle$$
$$\langle \textit{push}, c, Z, \textit{pop}, Z \rangle$$
$$\langle \textit{pop}, a, A, \textit{pop}, \lambda \rangle$$
$$\langle \textit{pop}, b, B, \textit{pop}, \lambda \rangle$$
$$\langle \textit{pop}, \lambda, Z_0, \textit{pop}, \lambda \rangle$$

Note que o APN irá parar (não terá instruções para seguir) se ele ler um *a* enquanto *B* estiver no topo da pilha. A seguinte tabela mostra as configurações de pilha sucessivas nos segmentos de cadeia sucessivos para entrada *abbcbba*. Quando *c* é lido, a pilha contém o reverso da cadeia lida antes; depois de ler *c*, o sistema retira (*pop*) da pilha quando o símbolo de entrada casa com o topo da pilha.

| cadeia | pilha |
|---|---|
| *abbcbba* | $Z_0$ |
| *bbcbba* | $AZ_0$ |
| *bcbba* | $BAZ_0$ |
| *cbba* | $BBAZ_0$ |
| *bba* | $BBAZ_0$ |
| *ba* | $BAZ_0$ |
| *a* | $AZ_0$ |
| | $Z_0$ |

Esta linguagem era particularmente fácil para um APN manipular porque o centro *c* informava ao autômato quando "voltar" e desmontar a pilha. No próximo exemplo, não será tão simples assim.

**3. Exemplo.** Considere a linguagem $\{ ww^R \mid w \in (a + b)^* \}$. Aqui, o plano de processamento precedente não funciona porque numa busca da esquerda para a direita a posição do centro da cadeia é desconhecida. Portanto, é necessário introduzir não determinismo no processamento da pilha, essencialmente através da "adivinhação" do centro da cadeia quando o centro tiver de ser encontrado (quando dois símbolos consecutivos são idênticos). Então substitui-se $\langle$ *push*, *c*, *Z*, *pop*, *Z* $\rangle$ no APN precedente pela regra-$\lambda$ (porque a "entrada" é apenas $\lambda$, não um símbolo de entrada) $\langle$ *push*, $\lambda$, *Z*, *pop*, *Z* $\rangle$ que diz que quando estiver o estado *push* o APN pode "escolher"

ir para o estado *pop* sem usar um símbolo de entrada ou alterar a pilha. Aqui está o dispositivo:

$\langle$ *push*, *a*, *Z*, *push*, *AZ* $\rangle$
$\langle$ *push*, *b*, *Z*, *push*, *BZ* $\rangle$
$\langle$ *push*, $\lambda$, *Z*, *pop*, *Z* $\rangle$
$\langle$ *pop*, *a*, *A*, *pop*, $\lambda$ $\rangle$
$\langle$ *pop*, *b*, *B*, *pop*, $\lambda$ $\rangle$
$\langle$ *pop*, $\lambda$, $Z_0$, *pop*, $\lambda$ $\rangle$

A máquina é *não determinística* porque quando estiver no estado *push* ela tem duas transições de estado possíveis: ou ler um símbolo de entrada *ou* mudar para o estado *pop*. Claramente, a cadeia é aceita pelo APN apenas se (i) a cadeia for da forma $ww^R$ *e* (ii) a transição para *pop* for feita exatamente na metade da leitura da cadeia.

Estamos usando o mesmo princípio usado para os autômatos não determinísticos da parte anterior: uma cadeia é aceita se existe *pelo menos uma* escolha das transições de estado possíveis que leva a alguma configuração de aceitação da máquina (na instância presente: pilha vazia depois da entrada ter sido lida nela inteiramente). Em geral, então, o processamento de autômato de pilha é acompanhado por um conjunto de quíntuplas da forma

(estado, símbolo-lido, topo-da-pilha, novo-estado, novo-topo-da-pilha)

Tais quíntuplas expressam como as transições entre descrições instantâneas da forma

(estado, entrada não lida, pilha)

são acompanhadas e na realidade representam uma linguagem de programação primitiva.

**4. Definição.** Um autômato de pilha (APN) (não determinístico) é uma séptupla $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, onde:

(1) $Q$ é um conjunto finito de *estados*;
(2) $\Sigma$ é um conjunto finito de símbolos de *entrada*;
(3) $\Gamma$ é um conjunto finito de símbolos de *pilha*;
(4) $\delta$ é o conjunto de transições $\langle q, x, Z, q', \sigma \rangle$ que também se escreve na notação $(q', \sigma) \in \delta(q, x, Z)$ tal que se possa considerar $\delta$ como uma *função de transição*:

$$\delta: Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \rightarrow \text{subconjuntos finitos de } Q \times \Gamma^*$$

(5) $q_0$ é o *estado inicial*;
(6) $Z_0$ é o *símbolo de pilha inicial*;
(7) $F \subset Q$ é um conjunto de *estados de aceitação*.

Dada esta maquinaria, podemos falar sobre uma *descrição instantânea* (*DI*) de um APN *M*. Uma DI para um APN é uma tripla ($q$, $w$, $\sigma$) onde $q$ é um estado, $w = x_1x_2...x_n$ é uma cadeia de símbolos de entrada ainda a ser lidos com o APN correntemente lendo $x_1$, e $\sigma = Z_1Z_2...Z_m$ é a cadeia de símbolos na pilha com $Z_1$ no topo e $Z_m$ no fundo.

As transições mapeiam DIs para DIs não-deterministicamente de duas formas: Se *M* estiver no estado $q$ com o símbolo de entrada corrente $x$ e elemento de pilha corrente $A$, então

(a) Se $\langle q, x, A, q', A_1 \ldots A_k \rangle$ para $x \in \Sigma$ é uma quíntupla do APN, então *M* pode (não determinismo!) causar a seguinte transição DI:

$$(q, xw, A\sigma) \Rightarrow (q', w, A_1 \ldots A_k\sigma);$$

(b) Se $\langle q, \lambda, A, q', A_1 \ldots A_k \rangle$ é uma quíntupla do APN, então sua execução pode causar a transição DI:

$$(q, xw, A\sigma) \Rightarrow (q', xw, A_1 \ldots A_k\sigma).$$

Isto é, em (b) a máquina processa a pilha sem avançar a entrada de nenhuma forma. Mantendo as convenções para gramáticas, usa-se o símbolo "$\Rightarrow^*$" no sentido *tripla* 1 $\Rightarrow^*$ *tripla* 2 para significar que a DI *tripla* 2 pode ser derivada depois de uma seqüência finita de zero ou mais transições a partir de *tripla* 1.

**5. Definição.** Define-se a *linguagem aceita pela pilha vazia por um APN M* como

$$T(M) = \{ w \mid (q_0, w, Z_0) \Rightarrow^* (q, \lambda, \lambda), \text{ para qualquer } q \in Q \}.$$

Isto é, a linguagem aceita pela pilha vazia por um APN é o conjunto de todas as cadeias que, quando completamente processadas, levam a uma pilha vazia. Note que *F não* é usado nesta definição. Entretanto, *podemos* dar uma outra definição baseada em *F* de aceitação por um APN, que veremos é equivalente a definição anterior.

**6. Definição.** Seja *M* um APN. Define-se *o conjunto de cadeias aceitas pelo estado final por M* como

$$F(M) = \{ w \mid (q_0, w, Z_0) \Rightarrow^* (q_a, \lambda, \sigma), \text{ para algum } q_a \in F \text{ e } \sigma \in \Gamma^* \}.$$

**7. Teorema.** *Uma linguagem L é APN aceitável pelo estado final se e somente se ela for APN aceitável pela pilha vazia*.

PROVA: Mostramos que $L = T(M)$ implica que existe um APN $M'$ tal que $L = F(M')$. Suponha que $L = T(M)$. Modifique *M* inserindo um novo símbolo de pilha $Z'$ abaixo do fundo de pilha usual $Z_0$, usando a quíntupla $\langle q_0, \lambda, Z_0, q_0, Z_0Z' \rangle$. Adicione um novo estado $q_a$, e faça *F*, o conjunto de estados finais, igual a $\{q_a\}$. Agora adicione a regra adicional $\langle q, \lambda, Z', q_a, Z' \rangle$ a *M*, para cada estado $q$. A máquina resultante aceitará *L* pelo estado final. ◊

**8. Definição.** Um autômato de pilha *generalizado* é um APN que se comporta de acordo com a descrição de máquina da Definição 4 exceto que podem existir transições em $\delta$ as quais lêem uma *cadeia* de símbolos de pilha (ao invés de exatamente um símbolo). Então, podem existir finitamente muitas quíntuplas da forma

$$\langle\, q, a, B_1 \ldots B_k, q', C_1 \ldots C_n \,\rangle$$

**9. Proposição.** *Se uma linguagem L é aceita por um APN generalizado, então L é aceita por um APN*.

Na Seção 3 desta parte provaremos que as linguagens livres de contexto são exatamente as linguagens aceitas por autômatos "push-down". Metade deste resultado, que mostra que toda linguagem livre de contexto é aceita por um autômato de pilha, é baseada na forma normal de Greibach para uma gramática livre de contexto. Estabelece-se este resultado de forma normal na Seção 2. Para motivar esta "simulação" de gramáticas livres de contexto por autômatos de pilha, primeiro mostra-se como o AND derivado de uma gramática linear a direita pode ser visto como um APN de um estado.

**10. Observação.** Dado um AND $M = (Q, \delta, Q_0, F)$, forme o APN de um estado $M\hat{}$. Usa-se o símbolo $*$ para denotar o estado único de $M\hat{}$. Os símbolos da pilha de $M\hat{}$ são os estados, $Q$, de $M$. Então podemos escrever $M\hat{}$ como:

$$M\hat{} = (\{*\}, \Sigma, Q, \delta\hat{}, *, q_0, \{*\})$$

as transições $\delta\hat{}$ de $M\hat{}$ imitam $\delta$ de acordo com a fórmula "trate o símbolo no topo da pilha de $M\hat{}$ como o estado de $M$," tal que

$$(*, q') \in \delta\hat{}(*, x, q) \quad \text{sse} \quad q' \in \delta(q, x)$$

É então claro que

$$(*, w, q_0) \Rightarrow^* (*, \lambda, q) \quad \text{sse} \quad q \in \delta^*(q_0, w)$$

Se adicionar-se a $\delta\hat{}$ as regras

$$(*, \lambda) \in \delta\hat{}(*, x, q) \quad \text{sse} \quad \delta(q, x) \in F$$

deduz-se que

$$(*, w, q_0) \Rightarrow^* (*, \lambda, \lambda) \quad \text{sse} \quad \delta^*(q_0, w) \in F.$$

Então

$$T(M\hat{}) = T(M).$$

Em termos de gramática linear a direita, isto diz que a regra

$$A \rightarrow bB$$

fornece a quíntupla $\langle *, b, A, *, B \rangle$ (que agora será abreviada para $\langle b, A, B \rangle$ como no Exemplo 1), enquanto a regra

$$A \to b$$

fornece a quíntupla $\langle *, b, A, *, \lambda \rangle$. Em outras palavras, agora usando $S$ no lugar de $q_0$, a derivação

$$S \Rightarrow^* w_1 A \Rightarrow w_1 bB \Rightarrow^* w_1 bw_2 = w$$

é imitada pelo APN passando através das DIs

$$(*, w, S) \Rightarrow^* (*, bw_2, A) \Rightarrow (*, w_2, B) \Rightarrow^* (*, \lambda, \lambda)$$

Informalmente, isto diz que a cadeia de entrada processada parcialmente contém a porção da cadeia original $w$ que não é lida e o total da cadeia é aceita sse a variável na pilha pode derivar esta cadeia.

O próximo exemplo generaliza este fato para um caso no qual a pilha contém uma *seqüência* de variáveis, ao invés de apenas uma.

**11. Exemplo.** Considere a gramática para $\{ a^n b^n \mid n \geq 1\}$ na forma normal de Greibach usando produções

$$S \to aSB \mid aB$$
$$B \to b$$

Define-se um APN *M não determinístico* de um estado com as transições

$$\langle a, S, SB \rangle$$
$$\langle a, S, B \rangle$$
$$\langle b, B, \lambda \rangle$$

Claramente, na leitura da cadeia de entrada $a^n b^n$ temos os resultados intermediários possíveis

(1) $(*, a^n b^n, S) \Rightarrow^* (*, a^{n-j} b^n, SB^j)$
(2) $(*, a^n b^n, S) \Rightarrow^* (*, a^{n-k} b^n, B^k)$ $(k > 1)$
(3) $(*, a^n b^n, S) \Rightarrow^* (*, b^n, B^n)$

A derivação (1) é um estágio intermediário para derivar uma DI da forma (2) ou (3). Mas (2) é um *dead end* - nenhuma transição é aplicável - se $n > k$, enquanto (3) está *no caminho* da derivação completa

$$(*, a^n b^n, S) \Rightarrow^* (*, \lambda, \lambda).$$

## 2.6. O Teorema da Equivalência

O *teorema da equivalência* estabelece que as linguagens aceitas por APNs são precisamente as linguagens livres de contexto.

**1. Teorema.** *Seja L = L(G) para alguma gramática livre de contexto G. Então L é aceita por algum APN (em geral, não determinístico). Ou seja, L = T(M) para algum APN.*

PROVA: Sem perda de generalidade, assuma que $G$ está na forma normal de Greibach. Devemos fornecer um autômato para $L(G)$ na forma de um APN de um estado $M$. Como $M$ tem apenas um estado $*$ podemos novamente abreviar quíntuplas $\langle *, x, z, *, w \rangle$ para a forma $\langle x, z, w \rangle$. Então associe qualquer regra da forma $A \rightarrow bCDE$ com a instrução APN $\langle b, A, CDE \rangle$. Qualquer regra da forma $A \rightarrow b$ leva à transição $\langle b, A, \lambda \rangle$. Regras-lambda tornam-se movimentos com nenhuma entrada: $A \rightarrow \lambda$ corresponde a $\langle \lambda, A, \lambda \rangle$. Devemos agora provar que $L = T(M)$. Para fazê-lo, provamos o resultado mais forte que

$$(*, w_1w_2, S) \Rightarrow^* (*, w_2, \sigma)$$

(isto é, depois de ler $w_1$, $M$ terá a cadeia $\sigma$ de símbolos em sua pilha) tem sentido sse $S \Rightarrow^* w_1\sigma$ é uma derivação válida em $G$. Pegando $w_2 = \sigma = \lambda$, vemos que a equivalência precedente implica que $(*, w_1, S) \Rightarrow^* (*, \lambda, \lambda)$ sse $S \Rightarrow^* w_1$ que diz que $w_1 \in T(M)$ sse $w_1 \in L(G)$. Provaremos agora nosso resultado mais forte por indução no comprimento de $w_1$.

*Base*: Para $w_1 = \lambda$ existem dois casos:
a) Temos ambos $(*, w_2, S) \Rightarrow^* (*, w_2, S)$ e o correspondente $S \Rightarrow^* S$.
b) Se $S \rightarrow \lambda$ é uma produção de $G$, temos ambos $(*, w_2, S) \Rightarrow^* (*, w_2, \lambda)$ e o correspondente $S \Rightarrow^* \lambda$.

*Indução*: Suponha que $w_1$ é tal que $(*, w_1w_2, S) \Rightarrow^* (*, w_2, \sigma)$ sse $(S \Rightarrow^* w_1\sigma)$. Por conveniência escrevemos $\sigma = A\tau$, $w_2 = aw_3$. Vamos checar agora a indução para $w_1a$. Como $(*, w_1aw_3, S) \Rightarrow^* (*, aw_3, A\tau)$ pela hipótese indutiva, temos $(*, w_1aw_3, S) \Rightarrow^* (*, w_3, \sigma'\tau)$ sse $S \Rightarrow^* w_1A\tau$ para algum $A$ com $A \rightarrow a\sigma'$, isto é, sse $S \Rightarrow^* w_1a\sigma'\tau$, que está provado. ◊

**2. Exemplo.** Considere a seguinte gramática na forma normal de Greibach para a linguagem dos parênteses casados:

$$S \rightarrow (L \mid \lambda$$
$$L \rightarrow (LL \mid )$$

O seguinte é uma derivação mais a esquerda da cadeia (()()) nesta gramática.

$$S \Rightarrow (L \Rightarrow ((LL \Rightarrow (()L \Rightarrow (()(LL \Rightarrow (()()L \Rightarrow (()())$$

Dá-se o APN associado (de um estado) a seguir, com o símbolo $S$ designando o fundo da pilha.

$$\langle (, S, L \rangle$$

$$\langle\ \lambda,\ S,\ \lambda\ \rangle$$
$$\langle\ (,\ L,\ LL\ \rangle$$
$$\langle\ ),\ L,\ \lambda\ \rangle$$

Para APNs de um estado pode-se reverter o método do resultado prévio para achar uma gramática livre de contexto que gere a linguagem aceita pelo APN.

**3. Exemplo.** Considere o Exemplo 1 da Seção 2.1, um APN de um estado que aceita a linguagem de números iguais de *a*'s e *b*'s. Sua gramática associada (substituindo $Z_0$ por $Z$) é

$$Z \rightarrow aAZ|bBZ|\lambda$$
$$A \rightarrow aAA|b$$
$$B \rightarrow bBB|a$$

Vamos resumir este processo estabelecendo o seguinte resultado:

**4. Teorema.** *Seja M um APN de um estado. Então existe uma gramática livre de contexto G tal que L(G) = T(M).*

PROVA: Para formar a gramática, converta triplas da forma $\langle a, B, \sigma\rangle$ para $\sigma \in \Gamma^*$ em produções da forma $B \rightarrow a\sigma$. Converta triplas da forma $\langle\ \lambda,\ B,\ \sigma\ \rangle$ em produções da forma $B \rightarrow \sigma$. A prova de L(G) = T(M) é similar a do Teorema 1. ◊

Por causa do Teorema 4, precisamos apenas provar que qualquer APN pode ser simulado por um APN de um estado a fim de estabelecer a equivalência completa de linguagens livres de contexto e a classe de linguagens aceitas pelos APNs. O próximo teorema estabelece este resultado.

**5. Teorema.** *Seja M um APN. Então existe um APN de um estado M' tal que T(M) = T(M').*

ESBOÇO DA PROVA: A idéia da prova é projetar a maquinaria de pilha de *M'* tal que *M'* possa simular o controle de estados finitos de *M* em sua pilha. Se *q* e *p* são estados e *A* é um símbolo da pilha de *M*, então a presença da composição de símbolos da pilha [*p A q*] na pilha de *M'* significa "*M* está no estado *p*, lendo um *A* no topo da pilha; quando *A* é apagado da pilha, *M* está no estado *q*."

Agora uma instrução da forma

($\alpha$) $\langle\ p,\ a,\ A,\ q,\ BCD\ \rangle$

em *M* é simulada em *M'* por *todas* as triplas

$$\langle\ a,\ [p\ A\ *_3],\ [q\ B\ *_1][*_1\ C\ *_2][*_2\ D\ *_3]\ \rangle$$

não importando a escolha dos símbolos $*_1$, $*_2$ e $*_3$ como estados de *M*. Ou seja, neste caso a quíntupla *M* original é substituída em *M'* por todas as triplas que produzem a mesma seqüência de símbolos de pilha e o mesmo estado inicial.

Um esquema similar é usado para simular as quíntuplas de *M* que lêem nenhuma entrada mas não encurtam a pilha. Para quíntuplas da forma

($\beta$) $\langle\ p,\ a,\ A,\ q,\ \lambda\ \rangle$

que retiram da pilha, entretanto, somos mais restritivos e estas regras são a chave para a construção. Simulamo-as com instruções da forma

$$\langle\ a, [p\ A\ q], \lambda\ \rangle,$$

isto é, quando instruções de redução da pilha são aplicadas, a pilha deve diminuir de acordo com o estado resultante de *M* e com o estado inicial.

Tratamos da condição inicial preprocessando $Z_0$, o símbolo inicial para *M'*, com o conjunto de instruções

$$\langle\ \lambda, Z_0, [q_0\ Z_0\ *]\ \rangle.$$

Dada uma DI de *M'*, digamos $(w, [p\ A_1\ p_1]...[p_{k-1}\ A_k\ p_k])$, referimos à tripla $\langle\ p, w, A_1...A_k\ \rangle$ como a *espinha* ("spine") da DI. A espinha de uma DI de *M'* é apenas a DI com todas as informações de estado exceto o estado do topo tirado. No caso de $k = 0$, definimos a espinha da DI $(w, \lambda)$ como $(-, w, \lambda)$.

Para mostrar $T(M) = T(M')$, devemos mostrar que $T(M) \subset T(M')$ e $T(M') \subset T(M)$. Vamos mostrar a primeira relação. Provamos que $w \in T(M)$ implica que $w \in T(M')$ mostrando que se $(q, w, \sigma)$ é uma DI de *M*, então uma DI de *M'* é produzida com a espinha $\langle\ q, w, \sigma\ \rangle$ ou, no caso de $\sigma$ ser vazio, com a espinha $\langle\ -, w, \lambda\ \rangle$. Claramente, se *w* é aceita por *M* , ela é então aceita por *M'*. Nossa prova é por indução no número de passos necessários para obter a DI de *M* $(q, w, \sigma)$.

Seja $(q, w, \sigma)$ uma DI de *M*. Se ela é produzida depois de um passo de *M*, então ou $\sigma = A_1...A_k$ com $k \neq 0$, em tal caso $(q, w, \sigma)$ é uma espinha para todas as DIs de *M'* da forma $(w, [q\ A_1\ p_1]...[p_{k-1}\ A_k\ p_k])$ para todas as possíveis combinações de estados $p_1,..., p_k$, ou então $\sigma$ é vazio. No último caso, a quíntupla

$$\langle\ q_0, a, Z_0, q, \lambda\ \rangle$$

é simulada em *M'* por

$$\langle\ a, [q_0\ Z_0\ q], \lambda\ \rangle$$

que, quando aplicada em *M'* leva à espinha correta.

Agora assuma por indução que se a DI de *M* $(p, aw, A\sigma)$ para $A \in \Gamma$, $\sigma \in \Gamma^*$ é gerável com *n* passos de APN por *M*, então $\langle\ p, aw, A\sigma\ \rangle$ é a espinha de alguma DI de *M'*. Suponha que uma instrução da forma $(\alpha)$ seja aplicável, levando à DI de *M* $(q, w, BCD\sigma)$ depois de $n + 1$ passos. Então como por indução uma espinha $\langle\ p, aw, A\sigma\ \rangle$ é derivável em *M'*, uma DI de *M'* com espinha $\langle\ q, w, BCD\sigma\ \rangle$ é também derivável.

Considere agora o caso de uma regra de apagamento: Suponha no passo $n + 1$ $(q, w, \sigma)$ seja produzida a partir de $(p, aw, A\sigma)$ onde $\sigma = B_1...B_k$. Então como a espinha $\langle\ p, aw, A\sigma\ \rangle$ eqüivale a todas as combinações de DIs de *M'* da forma

$$(aw, [p\ A\ *_1], [*_1\ B_1\ *_2]...[*_k\ B_k\ *_{k+1}])$$

então $*_1$ pode ser instanciado a *q* e a instrução

$$(a, [p\ A\ q], \lambda)$$

produz o efeito desejado. Isto completa a prova que $T(M) \subset T(M')$. $\Diamond$

Na Seção 1.3, mostramos que todo autômato finito não determinístico (AFN) *M* é equivalente a um autômato finito determinístico (AFD), *M'*, isto é, $T(M) = T(M')$. Nos exemplos 1 e 2 da Seção 2.1 usamos APNs determinísticos, mas no exemplo 3 da mesma seção nosso APN usava não determinismo para "escolher" quando mudar do estado *push* para o estado *pop*. Este não determinismo é necessário? Mais geralmente, toda LLC é aceitável por algum APN *determinístico*? A resposta é não: existem LLCs que não podem ser aceitas por APNs determinísticos.