



Rusumo Prova II

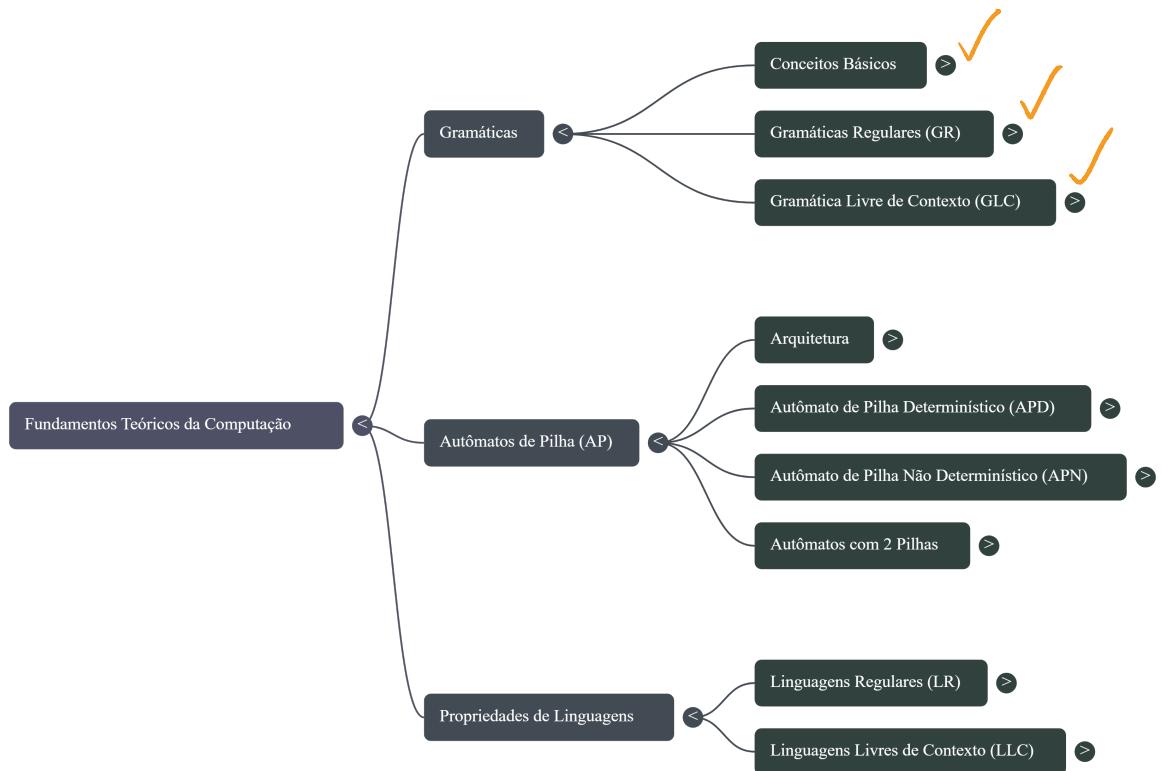
FTC

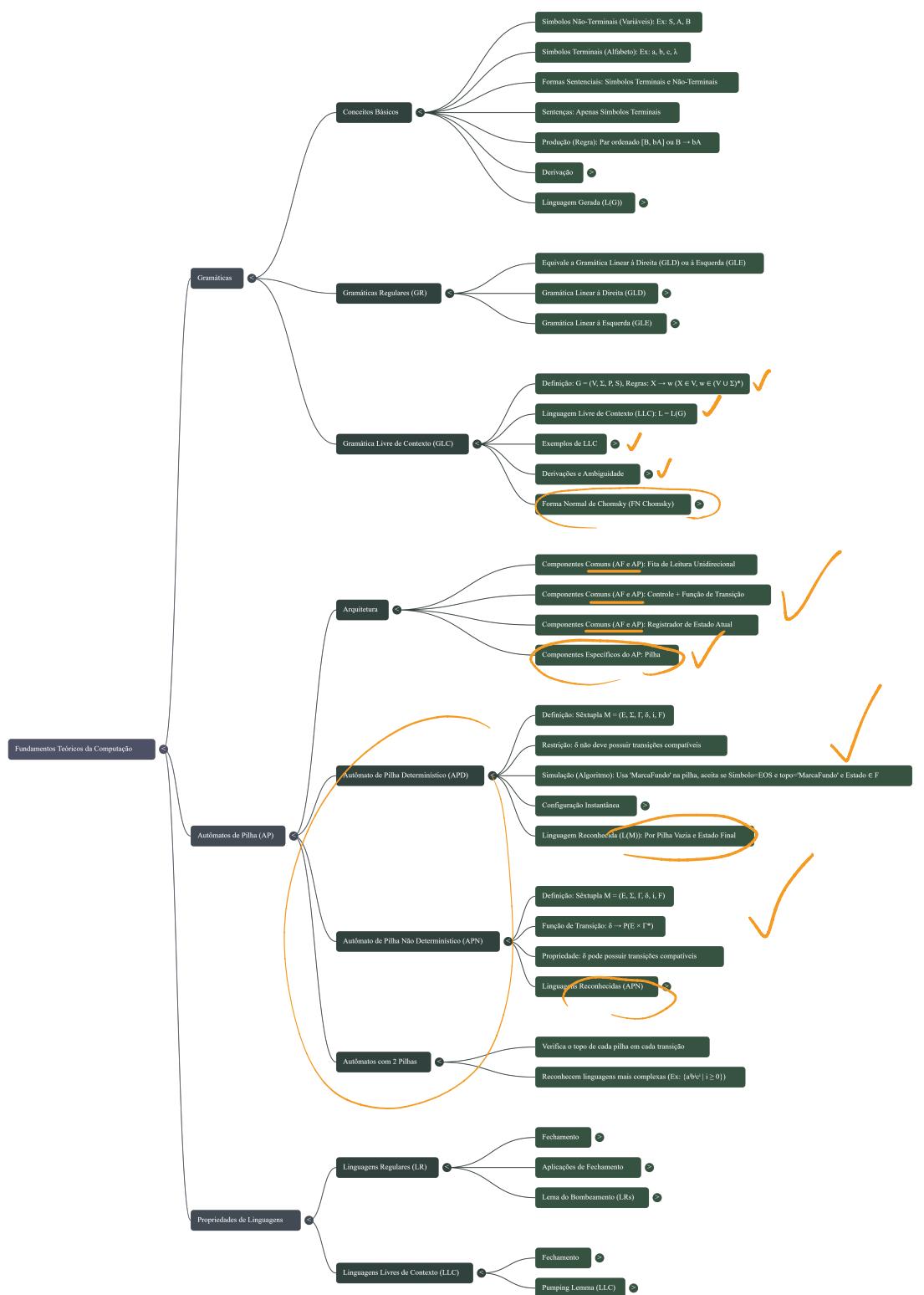
Sophia Carmazza



Materia:

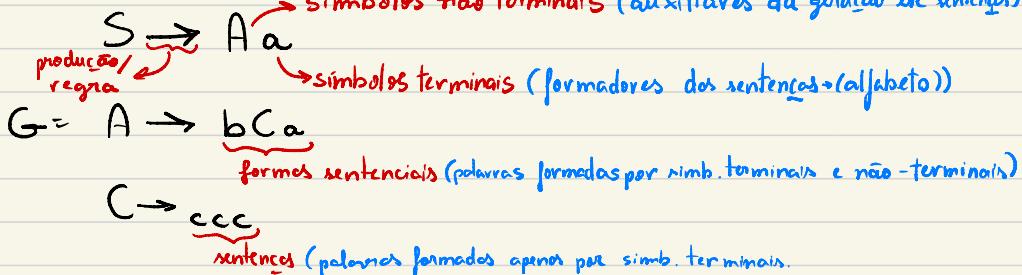
- gramática
- gramática e linguagem regular
- gramática e linguagem livre de contexto
- derivação (árvore de parser)
- ambiguidade de gramática (+ de 1 árvore de parser diferente pra mesma string)
- gerar gramática a partir de uma linguagem
- máquina/ autômato de pilha
- forma normal de chomsky
- lema do bombeamento





Gramática → usada p/ se especificar (gerar) uma linguagem

- * ex: gramática regular → linguagem regular



* **Derivação** → obtenção de uma forma sentencial ($b \underset{\curvearrowleft}{\overset{\curvearrowleft}{\longrightarrow}} c$) a partir de outra (ex: dado $B \rightarrow bA$ e cBc , derivemos: $cBc \rightarrow cbAc$)

$S \Rightarrow w$ significa que w é derivado a partir da variável S em um passo

$S \xrightarrow{*} w$ significa que w é derivado a partir da variável S em zero ou mais passos

* **composição da gramática** → $G = (V, \Sigma, P, S)$

- V = símbolos não terminais (variáveis)
- Σ = símbolos terminais (alfabeto, $V \cap \Sigma = \emptyset$)
- P = produções / regras
- S = símbolo inicial (variável de partida)

* **Linguagem gerada por uma gramática** → $L(G) = \{w \in \Sigma^* \mid S \xrightarrow{*} w\}$

Reclusão

Regras recursivas permitem que uma gramática gere infinitas palavras.

Reclusão

Direta	$: A \rightarrow aA$
Indireta	$: A \xrightarrow{*} w \xrightarrow{*} aA \quad (\text{e } w \text{ não contém } A)$

* (Seja $G = (V, \Sigma, P, S)$,
 $\{S \rightarrow AA, A \rightarrow AAA \mid bA \mid AbA\}, S\}$)

Derivação Mais a Direita (DMD) → $S \rightarrow AA \rightarrow Aa \rightarrow AAAa \rightarrow AAbAa \rightarrow AAbca \rightarrow AbAba \rightarrow AbAbaa$

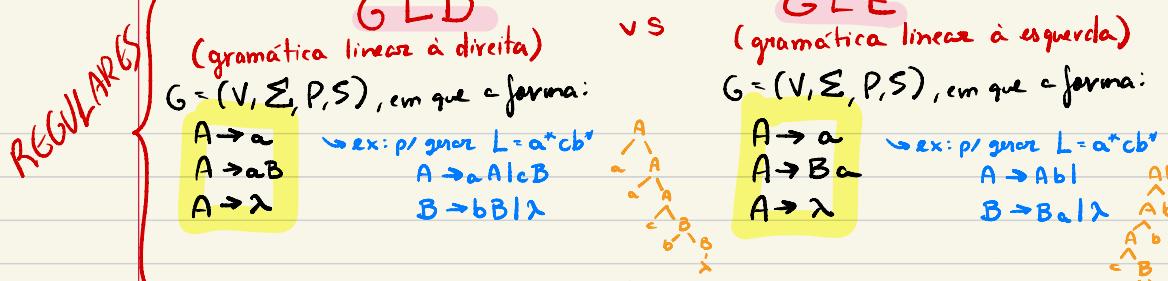
Durante o processo de derivação o não-terminal expandido de cada forma sentencial é sempre o mais a direita.

→ $Ababaa \rightarrow ababaa$

Derivação Mais a Esquerda (DME) → $S \rightarrow AA \rightarrow aA \rightarrow aAAA \rightarrow abAAA \rightarrow abcAA \rightarrow abcaAA$

Durante o processo de derivação o não-terminal expandido de cada forma sentencial é sempre o mais a esquerda.

→ $ababaA \rightarrow ababaa$



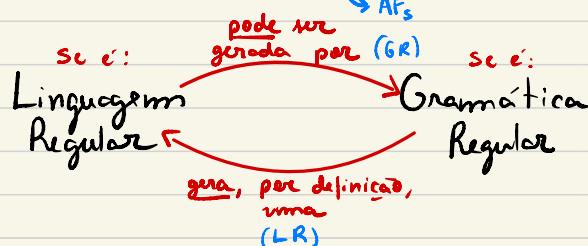
Gramática Regular → Nível 3 da Hierarquia de Chomsky
 É uma gramática regular se e só se é uma GLD ou GLE (se estiver em um desses formatos). Ela proíbe produções com recursão bilateral (ex: $P \rightarrow OPT$) ou múltiplos variáveis no lado direito (ex: $A \rightarrow BC$);

→ a principal característica que define uma GR é essa restrição estrutural (isso só é GR se for uma GLD ou uma GLE)

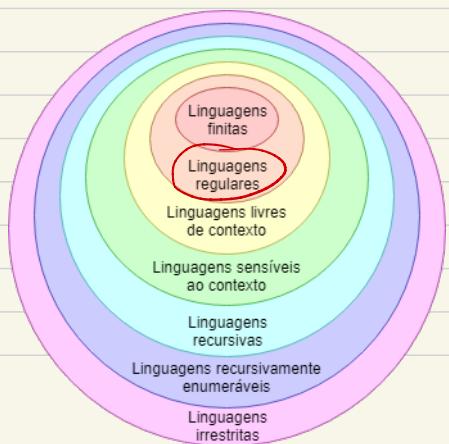
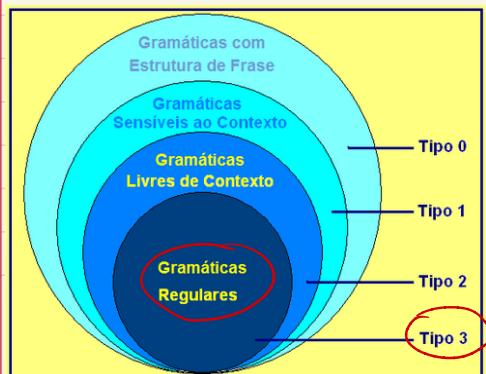
→ a gramática regular é uma forma de especificar as linguagens regulares.

→ é a classe menos poderosa na hierarquia de Chomsky, pq tem a restrição linear que garante a geração de linguagem regular (reconhecidos por Autômatos Finitos que têm memória limitada)

Ex: se a gramática precisa gerar uma contagem infinita ou empilhar/desempilhar simultaneamente, ela precisa de uma produção + complexa ↴



GLC = Gramática Livre de Contexto



e portanto não podem ser gerados por uma GR.

*Ex. de linguagens não regulares: linguagens que exigem balanceamento ou comparação de contagem, como:

- $L_1 = \{a^n b^n \mid n \geq 0\}$
- $L_2 = \{a^m b^n \mid m \leq n\}, L_3 = \{a^m b^n \mid m \geq n\}, L_4 = \{a^m b^n \mid m \neq n\}$
- $L_5 = \{ww \mid w \in \{a, b\}^*\}$
- $L_6 = \{w \in \{a, b\}^* \mid |a(w)| = n_b(w)\} \Rightarrow (\text{n de a's} = \text{n de b's})$
- $L_7 = \{a^K b^m c^n \mid K = m + n\} \Rightarrow \text{ soma de contagens}$

um n arbitrário

(variável)

ilimitado

Características de linguagens não-regulares:

1- contagem, balanceamento ou comparação (igualdade ou relações de ordem) = o AF autômato precisa contar e lembrar a qntd de um símbolo p/ compará-lo a outro ($=, >, <$, etc) e garantir alguma condição.

- ↳ ex: • $L = \{a^n b^n\} \Rightarrow$ (igualdade - precisa lembrar o valor de n)
- $L = \{a^m b^n \mid m \leq n\} \Rightarrow$ (ordem - precisa contar m e n)

2- linguagens de estruturas de padrões simétricos (repetições de palavras) = qualquer palavra que seja repetição de si mesma, pq ele não é capaz de lembrar a primeira metade (que pode ter tamanho ilimitado) p/ comparar c/ a segunda.

- ↳ ex: • $L = \{ww \mid w \in \{a, b\}^*\}$

3- linguagens baseadas em propriedades numéricas complexas (única) = se a propriedade do expoente é complexa e exige conhecimento infinito. O AF não consegue distinguir entre se n de símbolos grande o suficiente p/ ser um quadrado ou primo, por ex.

- ↳ ex: • $L = \{a^{n^2} \mid n \geq 0\} \Rightarrow$ quadrados perfeitos
- $L = \{a^n \mid n \text{ é primo}\} \Rightarrow$ ns primos

* Linguagens Regulares que parecem que não são:

1- Linguagens com restrição posicional fixa (algarismos)

= Quando o limite de "algarismos" for fixo, um infinito de estados é suficiente.

↳ ex: $L = \{w \in \{0,1\}^* \mid 3\text{º dígito de } w, \text{ da direita para esquerda, é } 1\}$

numérica ↑

2- Linguagens baseadas em módulo = Quando a restrição é verificada por um resto de divisão, é regular.

↳ ex: $L = \{w \in \{0,1\}^* \mid w \text{ representa um número binário divisível por } 6\}$

3- Linguagens que usam contagem de forma não-compositiva = Quando possui um n fixo p/ contagem ou ilimitado num dependência interna

↳ ex: $L = \{a^m b^n \mid m \text{ é par}\} \Rightarrow (a^{2k} b^*)$

Característica	Implica	Exemplo (Baseado em Fontes)
Memória Finita Suficiente (LR)	Regras fixas, propriedades modulares, padrões locais.	L_2 : Número binário divisível por 6 3.
Memória Infinita Necessária (LNR)	Relações de igualdade ou ordem entre contagens de símbolos, ou padrões globais de tamanho ilimitado.	$\{a^n b^n \mid n \geq 0\}$ 4; $\{a^m b^n \mid m \neq n\}$ 9; $\{ww \mid w \in \{a,b\}^*\}$ 11.

* Lema do Bombeamento p/ LR → no resumo anterior

Gramática Livre de Contexto → permite recursão

não restrita no lado direito da produção, desde que haja apenas um símbolo não-terminal no lado esquerdo.

↳ ex: $P \rightarrow OPL$

↳ não permite a geração de estruturas aninhadas ou balanceadas

* A aplicação da regra em X não depende de que está ao redor de X , apenas do próprio X (livre de contexto)

* Uma GLC é uma quadrupla $G = (V, \Sigma, P, S)$

↳ Toda regra $r \in P$ é da forma $X \rightarrow uu$, onde X é um único símbolo não-terminal ($X \in V$) e uu é uma string de qualquer combinação de variáveis e terminais (incluindo λ)



> Linguagem Livre de Contexto (LhCs) → tratam questões típicas de linguagens de programação, e contém a classe dos LR_s.

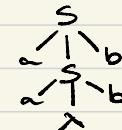
↳ Pode ser representada por um Autômato de Pilha ou por uma Gramática L.C. (GLC)

* Árvores de Derivação (de uma forma sentencial G): é uma árvore ordenada construída recursivamente assim:

- 1 uma árvore sem arestas cujo único vértice tem rótulo S é uma AD de S ;
- 2 se $X \in V$ é rótulo de uma folha f de uma AD A , então:
 - se $X \rightarrow \lambda \in P$, logo a árvore obtida acrescentando-se a A mais um vértice v com rótulo λ e uma aresta $\{f, v\}$ é uma AD;
 - se $X \rightarrow x_1 x_2 \dots x_n \in P$, em que $x_1, x_2, \dots, x_n \in V \cup \Sigma$, logo a árvore obtida acrescentando-se a A mais n vértices v_1, v_2, \dots, v_n com rótulos x_1, x_2, \dots, x_n , nesta ordem, e n arestas $\{f, v_1\}, \{f, v_2\}, \dots, \{f, v_n\}$ é uma AD.

* se a sequência das rótulos da fronteira de AD é igual a forma sentencial w , AD é uma árvore de derivação de w .

↳ ex: aa bb



Ambiguidade de Gramática

↳ Uma GLC é dita ambígua quando existe + de 1 Árvore de Derivação (AD) p/ alguma palavra / sentença gerada por G.
(ou seja, se tem várias formas de atribuir uma estrutura a uma mesma palavra).

$\neq AD_s \rightarrow \neq$ Estruturas Sintáticos $\rightarrow \neq$ Significados p/ a mesma string

↳ Se a gramática é ambígua, o analisador sintático não consegue determinar qual operação deve ser executada primeiro, resultando em estruturas semânticas conflitantes.



* Uma GLC é ambígua se, e somente se:

- 1- Existe + de 1 AD (árvore de derivação)
- 2- Existe + de 1 DMD (derivação + à direita)
- 3- Existe + de 1 DME (derivação + à esquerda)

pq essas duas não é uma forma de AD

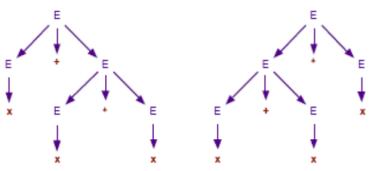
↳ Forma + comum de provar ambiguidade \Rightarrow encontrar 2 ADs p/ mesma string!



↳ Existe uma única DMD e DME (iguais!) correspondente a uma AD. Se conseguirmos construir duas ADs c/ estruturas \neq , as DMs podem ou não continuar sendo as mesmas.



♦ Exemplo: $x+x*x$



nem sempre é!

pode ser gerada

por pelo menos

uma G. ambígua

mas pode

parar em

gramáticas

não-ambíguas

- $E \Rightarrow E+E \Rightarrow X+E \Rightarrow x+E*x \Rightarrow x+x*x \Rightarrow x+x*x$
- $E \Rightarrow E+E \Rightarrow E+E*x \Rightarrow E+E*x \Rightarrow x+E*x \Rightarrow x+x*x \Rightarrow x+x*x$
- $E \Rightarrow E+E \Rightarrow E+E*x \Rightarrow X+E*x \Rightarrow x+E*x \Rightarrow x+x*x \Rightarrow x+x*x$
- etc...

- derivação mais à esquerda

$$\left. \begin{array}{l} E \Rightarrow E+E \Rightarrow x+E \Rightarrow x+E*x \Rightarrow x+x*x \Rightarrow x+x*x \\ E \Rightarrow E*x \Rightarrow E+E*x \Rightarrow x+E*x \Rightarrow x+x*x \Rightarrow x+x*x \end{array} \right\} 2x$$

- derivação mais à direita

$$\left. \begin{array}{l} E \Rightarrow E+E \Rightarrow E+E*x \Rightarrow E+E*x \Rightarrow x+E*x \Rightarrow x+x*x \Rightarrow x+x*x \\ E \Rightarrow E*x \Rightarrow E+E*x \Rightarrow E+E*x \Rightarrow x+E*x \Rightarrow x+x*x \Rightarrow x+x*x \end{array} \right\} 2x$$

(Deveria ter só 1 de cada)

Gramática Ambígua → Linguagem não-ambígua
não ambiguidade independe da entranha

★ Linguagens inerentemente ambíguas: Qualquer gramática que a gere é ambígua. É impossível achar uma GLC que a gere nem ambiguidade.

ex: $L = \{a^n b^m c^n | n, m \geq 0\} \cup \{a^n b^n c^m | n, m \geq 0\}$

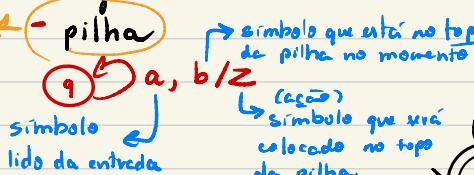
> Autômato de Pilha → reconhece LHCs e possui componentes adicionais, sendo capaz de ter uma memória limitada.

- fita de leitura unidirecional
- controle + função de transição
- registrador de estado atual

$$M = (E, \Sigma, I, \delta, i, F)$$

atados de alfabeto alfabeto da pilha estado inicial conjunto de estados finais

específico do AP



ex: $\{a^n b^n a^m | n, m \geq 0\}$

* se $a = \lambda$ → transições espontâneas, não precisa de input

* se $z = \lambda$ → representa "desempilhar" o símbolo b

* se $b = \lambda$ → transição ocorre num consulta à pilha (nada acontece)

* se $b = \lambda$ e $z \neq \lambda$ → representa "empilhar" o símbolo z .

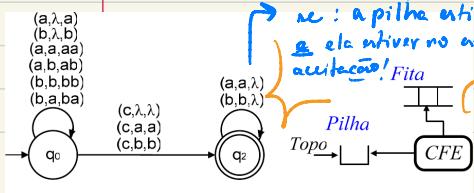
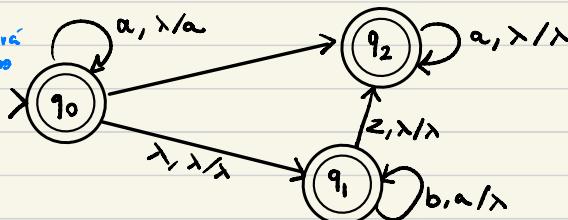
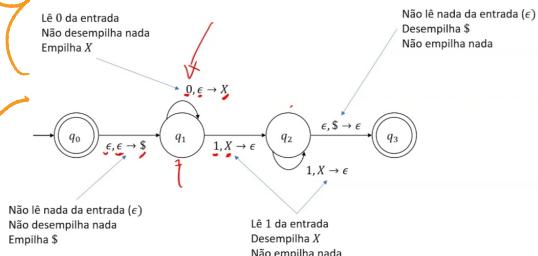


Diagrama de estados



Autômato de Pilha Não-Determinístico (APN)

vs

Autômato de Pilha Determinístico (APD)

- + poderoso → reconhece todos os LLCS e geralmente é o comum AP "normal"
- A função de transição mapeia um conjunto de pares de próx. estados e cadeias a empilhar.
- Ele pode ter transições compatíveis ou não. Se houver compatibilidade, ele "ramifica" a computação p/ seguir todos cam. possíveis.

$$L = \{0^m 1^n \mid m \geq n\}$$

$$L = \{w \in \{0, 1\}^* \mid w = w^R\}$$

Toda L reconhecida por um APN → LLC, e vice-versa!

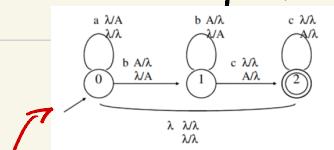
★ Transição Compatíveis → Quando, em um mesmo estado, múltiplos negros se aplicam à mesma entrada ou topo de pilha (incluindo o uso de λ). → eles criam ambiguidade sobre qual caminho p/ 2 trans. serem compatíveis, devem cumprir! → AP deve seguir.

deve cumprir ambos!

- {
 - ↳ condição de entrada e compatível ($a=c$ ou $a=\lambda$ ou $c=\lambda$) } p/ 1:
 e, a/b
 e, c/d

Autômatos de 2 pilhas:

→ Em cada transição o autômato verifica o símbolo no topo de cada pilha e executa uma op. em cada.



★ Nem toda L reconhecida por um AP de 2 pilhas é LLC.

Fechamento de Operações p/ LR_s

→ seja uma classe de linguagens L e uma operação sobre linguagens O . Diz-se que L é fechada sob O se a aplicação de O a linguagens pertencentes a L sempre resulta em uma linguagem pertencente a L .

analogamente

Considere duas linguagens regulares L_1 e L_2 , então:

- $L_1 \cup L_2$ também é regular (+) "ou"
- $L_1 \cap L_2$ também é regular (x) "e"

- $L_1 L_2$ também é regular
- L_1^* também é regular
- $\overline{L_1}$ também é regular

concatenação

repetição

complemento

Fechamento de Operações p/ LLC's

→ se L_1 e L_2 são LLC's, o resultado das operações abaixo também será uma LLC.

➤ Teorema: se L_1 e L_2 são linguagens livres do contexto, então:

- $L_1 \cup L_2$, $(L_1)^*$, $L_1 L_2$ são LLC, mas
(união) (fecho de (concatenação) Klore)

LLCs são fechados sob.
umas operações básicas =

- união
- fecho de Kleene
- concatenação

➤ Teorema: o conjunto das LLC's não é fechado sob interseção ou complemento:

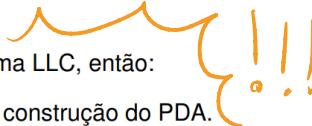
1. $L_1 \cap L_2$ pode não ser LLC - $a^i b^k \cap a^k b^k c^i = a^k b^k c^k$ $i, k > 0$.
(intersecção)

LLCs não
são fechados sob
uma operação

- intersecção
- complemento

2. $(L_1)'$ pode não ser LLC. Prova:

- considere L_1 e L_2 LLC. Faça $L = ((L_1)' \cup (L_2)')'$. Note que o resultado contradiz (1). $\rightarrow \overline{L_1 \cup L_2} = \overline{L_1} \cap \overline{L_2} \Rightarrow$ pode não ser LLC!
- $L = \{ ww \mid w \in \{a, b\}^*\}$ não é LLC. Mas L' é LLC.



➤ Teorema: seja L_1 uma LR e L_2 uma LLC, então:

- $L_1 \cap L_2$ é uma LLC. Prova por construção do PDA.

$L_1 \setminus L_2$
ou
 $L_1 - L_2$

$\{w \mid w \in L_1 \text{ e } \notin L_2\}$

$$L_1(LR) \cap L_2(LLC) \Rightarrow L_3(LLC)$$

(intersecção)

> A diferença de duas LLC's nem sempre são LLC tb!

Forma Normal de Chomsky

- é um formato padronizado e restrito p/ GLCs.
- da suje p/ simplificar e padronizar as regras (utiliza remove produção, vagas, derivação direcionada, etc)
- p/ análise algorítmica e provas de propriedades das LLGs) (pumping lemma)

> Uma GLC está na FNC se todas as suas produções pertencem a um desses formatos:

$$\left. \begin{array}{l} S \rightarrow \lambda \\ A \rightarrow a \\ A \rightarrow BC \end{array} \right\} \text{toda árvore de parse é uma árvore binária}$$

> Passo a Passo p/ transformar uma G em FNC:

1- Eliminar a recursão do símbolo inicial (garante que $S \neq \lambda$)
 → não recursivo → adiciona o novo inicial $S' \rightarrow S$ ou $S' \rightarrow S/\lambda$
 (se S for nulaável) ↗

2- Eliminar as transições lambda (remove produção da forma

$$A \rightarrow \lambda \quad (\text{exceto } S' \rightarrow \lambda, \text{ & excluir})$$

- 3- gerar gramáticos → 1- identificar todos os variáveis que geram λ .
 essencialmente não-contráctiles (λ tratados) → 2- marcar as variáveis que podem gerar λ indiretamente (via outros variáveis marcados)
 → 3- criar as novas produções nem as produções λ (quando todas as combinações permitem nem umas variáveis)
 → 4- eliminar o λ original.

→ produções unitárias!

3- Eliminar as regras de cadeias (remove todas as regras de produção da forma $A \rightarrow B$): (A pode derivar uma string u indiretamente)

(eliminar as regras de cadeias que não dependem de outras regras de cadeia) → seja $A \rightarrow B$ uma regra de cadeia → $A \rightarrow B \rightarrow u$, deve se add. a regra $A \rightarrow u$ ao conj. de produções

Exemplo:

$$G: \left\{ \begin{array}{l} S \rightarrow A \mid b \\ A \rightarrow B \mid a \\ B \rightarrow S \mid b \end{array} \right. \quad \begin{array}{l} \text{produções unitárias:} \\ S \rightarrow A \\ A \rightarrow B \\ B \rightarrow S \end{array}$$

* passo a passo detalhado de eliminação de regra de calic:

1º → construímos um fecho unitário p/ cada variável

$$\text{Fecho}(S) = \{S, A, B\}$$

$$\text{Fecho}(A) = \{A, B, S\}$$

$$\text{Fecho}(B) = \{B, S, A\}$$

\hookrightarrow

$\text{Fecho}(S)$ = todos variáveis que S pode derivar via regras diretas (indiretamente)

eliminaremos

2º os de fecho

menor, que depen-

dem de mais

de recursos e

complexidade no

processo

2º → substituir os produções unitárias

• p/ eliminar S $\rightarrow A$, substituir A por todas as produções de A que não são unitárias ($A \rightarrow a$).

• como S tb alcance B, adicionemos tb as produções não-unitárias de B ($B \rightarrow b$). → ficando $S \rightarrow a/b/b/b \Rightarrow \begin{cases} S \rightarrow a/b \\ A \rightarrow a/b \\ B \rightarrow a/b \end{cases}$

• fazemos o mesmo p/ A e B $\Rightarrow \begin{cases} A \rightarrow a/b \\ B \rightarrow a/b \end{cases}$

4- Eliminar os símbolos inúteis (remove símbolos que não alcancavam a partir do símbolo inicial ou que não derivam uma string de terminais)

A ORDEM
IMPORТА!!!

1º conj. que geram = { " , " , " }

2º indiretamente:

Se a variável tem só produz c/ recursos pra mesma e/ou produz outras que só jogam isso (nenhuma produz uma que gera terminal) \Rightarrow não gera terminal!
mas se produz uma que gera terminal (sem chamar outras que não geram junto)

remove os que
não estão
aqui!

1º removemos os símbolos que não geram terminais
2º removemos os símbolos não alcancáveis
removemos todos os produções que envolvem as variáveis removidas

conj. dos alcancáveis = { " , " , " }
pelos símbolos iniciais (removendo que não estavam aqui)

Terminals não podem aparecer multiplas c/ não-terminals ou fun+de 1 terminal pq as regras na FNC são limitadas a $A \rightarrow a$

5- Isolar terminals (p/ cada produção $A \rightarrow w$, onde w contém terminal a, e p/ cada terminal a com w):

→ cria-se uma nova produção $X_a \rightarrow a$ (não existe)

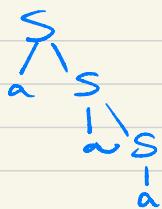
→ substituir o terminal a pelo novo não-terminal X_a em w.

$R_1 : S \rightarrow aABC$	$S \rightarrow X_aABC$
$R_2 : S \rightarrow a$	$S \rightarrow a$ (FNC)
$R_3 : A \rightarrow aA$	$A \rightarrow X_aA$
$R_4 : A \rightarrow a$	$A \rightarrow a$ (FNC)
$R_5 : B \rightarrow bbB$	$B \rightarrow X_bX_bB$
$R_6 : B \rightarrow bc$	$B \rightarrow X_bX_c$
$R_7 : C \rightarrow ccC$	$C \rightarrow X_cC$
$R_8 : C \rightarrow c$	$C \rightarrow c$ (FNC)

6 - Quebrar cadeias longas (p/ cada produção $A \rightarrow w$, onde o lado direito w tem comprimento maior que 2 ($|w| > 2$)):

- essa produção é substituída por uma sequência de produções binárias (criados agora)
- se $w = Bw' \Rightarrow$ a produção é substituída por:

$$\begin{array}{c} A \rightarrow BT \\ T \rightarrow w' \end{array}$$
um novo não-terminal



Forma Final (Exemplo):

Inicial:

$$S \rightarrow aABC1a$$

$$\begin{array}{l} G: A \rightarrow aA1a \\ B \rightarrow bbbB1bc \\ C \rightarrow cC1c \end{array}$$



passos 1, 2, 3 e 4
não são necessários
para uma G.



passo 5:

$$\begin{array}{l} S \rightarrow X_aABC1a \\ A \rightarrow X_aA1a \\ B \rightarrow X_bX_bB1X_bX_c \\ C \rightarrow X_cC1c \end{array}$$

novos não-terminais
não terminais
não terminais

passo 6:

$$\begin{array}{l} S \rightarrow X_aT_1T_21a \\ A \rightarrow X_aA1a \\ B \rightarrow X_bT_3B1X_bX_c \\ C \rightarrow X_cC1c \\ X_a \rightarrow a \\ X_b \rightarrow b \\ X_c \rightarrow c \\ T_1 \rightarrow AT_2 \\ T_2 \rightarrow BC \\ T_3 \rightarrow X_bB \end{array}$$

Final (FNC):

$$\begin{array}{l} S \rightarrow X_aT_1T_21a \\ A \rightarrow X_aA1a \\ B \rightarrow X_bT_3B1X_bX_c \\ C \rightarrow X_cC1c \\ X_a \rightarrow a \\ X_b \rightarrow b \\ X_c \rightarrow c \\ T_1 \rightarrow AT_2 \\ T_2 \rightarrow BC \\ T_3 \rightarrow X_bB \end{array}$$

0 Lema do Bombeamento p/ LHC garante que um símbolo não-terminal se repete em algum caminho (variável) na árvore de derivação longa.

Lema do Bombeamento (pumping lemma)

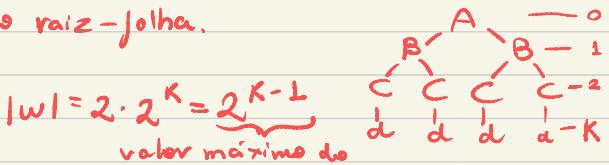
↳ ferramenta teórica usada p/ provar que uma determinada linguagem não pertence à classe dos LLGs.

> Seja L uma linguagem livre de Contexto gerada por uma gramática G que está na FNC, e seja K o número de símbolos não-terminal de G :

↳ Se existir uma string $w \in L$ tal que seu comprimento $|w| > 2^{K-1}$, então w pode ser decomposto na forma $w = p q v t y$, de modo que as seguintes condições sejam verdadeiras p/ p, q, v, t, y :

- 1) $w = p q v t y$
- 2) $|q v t| \leq 2^{K-1}$ → (comprimento da seção bombeável)
- 3) $|q| + |t| > 0$ → (garante que q e t contêm pelo menos 1 terminal)
- 4) $w_i = p q^i v t^i y \in L(G), \forall i \geq 0$

como a altura da árvore é maior que o m^o total de não-terminal, deve haver pelo menos um símbolo não-terminal que se repete no maior caminho raiz-folha.



Como a FNC Justifica a Aplicação do Lema:

A FNC é indispensável porque impõe uma restrição geométrica à Árvore de Derivação (AD) que permite a identificação de repetições:

1. **Estrutura Binária:** Como a gramática G está na FNC, toda árvore de parse (T) gerada por G é uma árvore binária.

2. **Relação Altura/Comprimento:** Em uma árvore binária gerada por uma gramática em FNC com altura n , o número de folhas (ou seja, o comprimento da string w) é, no máximo, $2^n - 1$, ou seja, $|w| \leq 2^n - 1$.

3. **Garantia de Repetição:** Se escolhermos uma string z longa o suficiente, tal que seu comprimento $|z|$ seja maior que $2^K - 1$ (onde K é o número de não-terminal), sua árvore de parse deve ter altura $n \geq K + 1$.

4. **O Princípio da Casa dos Pombos:** Como a altura da árvore é maior do que o número total de não-terminal (K), deve haver pelo menos um símbolo não-terminal que se repete no caminho de maior tamanho que parte da raiz e chega a alguma folha.

5. **A Ciclo de Produção ($A \rightarrow vAx$):** Essa repetição implica que existe um não-terminal A que deriva recursivamente a si mesmo ($A \rightarrow vAx$), permitindo que a parte intermediária (v e x) seja omitida ($i=0$) ou repetida i vezes ($i \geq 1$), gerando novos strings que permanecem em L .

comprimento de w, pq é uma árvore binária!

* Depois de definir uma palavra w que cumpre os requisitos,

↳ Testamos todos os casos/possibilidades de divisões em 5 partes.

↳ o lema só é satisfeito quando pelo menos uma divisão gera uma decomposição válida → (e não que todas dividam funcionem)

> Aplicações do Lema - Prova por Contradição

Asumir a Hipótese \rightarrow assumimos que a linguagem L é LLC.

\star Isto implica que deve existir um $n \in K$ tal que todos os strings w em L com $|w| \geq 2^{K-1}$ ponham em desordem reafirmando a LLC.

$$\star L = \{a^i b^i c^i \mid i \geq 0\}$$

\hookrightarrow Suponha que L é uma LLC gerada por uma GLC com FNC com K símbolos não terminais.

\hookrightarrow Tome $w = a^{2^{K-1}} b^{2^{K-1}} c^{2^{K-1}}$. Como $|w| = 3 \cdot 2^K$, $|w| > 2^{K-1}$, então

assim, pelo lema do bombeamento:

$$1) w = pqvty$$

$$2) |qv| \leq 2^{K-1}$$

$$3) |q| + |t| > 0$$

$$4) w_i = p q^i v t^i y \in L(G), \forall i \geq 0$$

1º CASO: $|a^{2^K}| b^{2^K} c^{2^K}$

get term so L
terminal

$$P \mid qvt \quad y$$

2º CASO: $|a^{2^K} ab| b^{2^K} c^{2^K}$

get term 2
terminal

$$P \mid q \quad | v \quad t \quad y$$

\Downarrow

bombeando a^i , $+a^j$ mas
= b^i e c^i , logo $w_i \notin L$

$a^{2^{K-2}} | a^i | ab | b^i | b^{2^{K-2}} c^{2^{K-2}}$

$P \quad q \quad v \quad t \quad y$

3º CASO: $|a^{2^K} b^{2^K} c^{2^K}|$

get term 2 terminal
simultaneously

$P \mid qvt \quad y$

\Downarrow

= bombeando a^i e b^i , $+a^j$ (b's mas = c's)
= análogo p/ b^i e c^i 's
= impossível p/ a^i e c^i

3º CASO: $|a^{2^K} b^{2^K} c^{2^K}|$

get term 2 terminal
simultaneously

$P \mid qvt \quad y$

\Downarrow

= bombeando perdendo a
ordem, logo $w_i \notin L$
 $\hookrightarrow ababab\dots$

4º CASO: $|a^{2^K} b^{2^K} c^{2^K}|$

get contém 3 terminais

$P \mid qvt \quad y$

\Downarrow

= impossível! pois $|qv| \leq 2^{K-1}$, em
que contém a^i , b^i e c^i ,
que $\geq 2^{K-1} + 2$

PROVA POR
(por contradição) = ABSURDO!

$$\begin{array}{l} FNC: \\ S \rightarrow \lambda \\ A \rightarrow - \\ A \rightarrow BC \end{array}$$

Segundo Exemplo:

$$L(G) = \{a^n b^n \mid n \geq 0\}$$

$$G: S \rightarrow aSb \mid \lambda$$

1. ↓ unir S'

$$G_1: S' \rightarrow S \mid \lambda$$

$$S \rightarrow aSb \mid \lambda$$

2. ↓ eliminar λ (exato) inicial

$$G_2: S' \rightarrow S \mid \lambda$$

$$S \rightarrow aSb \mid ab$$

(regras de cédria)

3. eliminar produções unitárias
(ex: $S' \rightarrow S$)

$$G_3: S' \rightarrow aSb \mid ab \mid \lambda$$

$$S \rightarrow aSb \mid ab$$

4. ↓ indicar terminal

$$G_4: S' \rightarrow X_a S X_b \mid X_a X_b \mid \lambda$$

$$S \rightarrow X_a S X_b \mid X_a X_b$$

$$X_a \rightarrow a$$

$$X_b \rightarrow b$$

5. ↓ quebrar cédrios longos (não-binários)

não temos inutéis nessa

não podemos mutar var terminais c/ não tem. de fator de L termini!

Aplicações do L.B.:

Se $L \in LL\backslash C$, existe \Leftarrow

um p tal que toda string $w \in L$ com $|w| > p$ pode ser dividida em 5 partes

$w = p q v t y$, satisfaçõe:

$$1) |qv| + |t| \leq p$$

$$2) |q| + |t| > 0 \quad (\text{em } n \geq 1)$$

$$3) w_i = p q^i v t^i y \in L(G), \forall i \geq 0$$

$$G_5: S' \rightarrow X_a T_L \mid X_a X_b \mid \lambda$$

(não importa ordem)

$$S \rightarrow X_a T_L \mid X_a X_b$$

$$X_a \rightarrow a$$

$$X_b \rightarrow b$$

$$T_L \rightarrow S X_b$$

sendo que

$$p = 2^{k-1}, \text{ sendo } k \text{ o n. de variáveis de } G.$$

escolhendo: $w = a^{2^k} b^{2^k}$

1º CASO: $a^{2^k} \mid b^{2^k}$ OU $a^{2^k} \mid b^{2^k} \mid y$

= bombeando, on de a's +, mas de b's =. $w_i \notin L(G)$.

2º CASO: $a^{2^k} b^{2^k} \mid y$ = bombeando $a \in L(G)$ com $a \in b$ e removendo abertos \hookrightarrow a ordenação inicial

3º CASO: $a^L \mid a^{\frac{2^k}{2}-2} \mid ab \mid b^{\frac{2^k}{2}-2} \mid b^L$ = bombeando ambos, $q \in t$, w_i mantém sua integridade ao manter a proporção igual de a's e b's.

* dividir por 2 garante que os letros estão divididos igualmente.

$$L(G) \in LL\backslash C !!$$

$$w_i \in L(G).$$

(separar qvt no meio da string)

Gabi: ↗

Tabela de Classificação de Linguagens Formais

I. Linguagens Regulares (LR) — Memória Finita (AF)

#	Linguagem (L)	Descrição Matemática	Classificação	Critério Chave	Confusão Comum na Prova
1	Sequências simples	$er = a^* b^* c^*$ $a^* b^* c^*$ → qualquer qntd. de cada um deus.	 A : $er = a^* b^* c^*$ B : $a^* b^* c^*$ → qualquer qntd. de cada um deus. C : $er = a^* b^* c^*$	LR Conhecida por AF. Não exige pareamento ou contagem relativa entre blocos.	Confundida com $a^n b^n$ se a contagem não for exigida.
2	Divisibilidade	$er = (0 \cdot 1)^*$ $L = \{w \in \{0, 1\}^* \mid w \text{ é divisível por } 6\}$ restos possíveis: $0, 1, 2, 3, 4, 5$ Só aceitamos resto 0	 A : $er = (0 \cdot 1)^*$ B : $L = \{w \in \{0, 1\}^* \mid w \text{ é divisível por } 6\}$ C : restos possíveis: $0, 1, 2, 3, 4, 5$ Só aceitamos resto 0	A divisibilidade pode ser verificada usando estados que <u>registram o resto da divisão</u> (memória finita).	Condições aritméticas finitas são geralmente LR.

II. Linguagens Livres de Contexto (LLC) — Memória de Pilha (AP)

A. LLC Determinísticas (LAD)

#	Linguagem (L)	Descrição Matemática	Classificação	Critério Chave	Confusão Comum na Prova
3	Pareamento Simples	 A : $a, \lambda/A$ B : $b, A/\lambda$ $L = \{0^n 1^n \mid n \geq 0\}$	LLC (APD)	Exige contagem e pareamento ; impossível para AF. Reconhecida por APD, pois a transição de empilhar 0s para desempilhar 1s é determinística (ocorre ao ler o primeiro 1).	É o exemplo clássico de não-LR.
4	Contagem Comparativa	 A : $a, \lambda/A$ B : $b, A/\lambda$ $m > n: S \rightarrow aSb\lambda A\lambda a$ $A \rightarrow Aa$ $m < n: S \rightarrow aSb\lambda A\lambda b$ $B \rightarrow Bb$ $m \leq n: S \rightarrow aSb\lambda a\lambda b$ $B \rightarrow Bb$	LLC (APD)	Requer que o número de 0s seja menor ou igual ao de 1s. Pode ser resolvida por um APD que empilha todos os 0s e verifica se a pilha está vazia antes da fita de 1s.	Provada Não-LR pelo Lema do Bombeamento para LRs.
5	Contagem com Marcador	 A : $a, \lambda/A$ B : $b, A/\lambda$ $m > n: S \rightarrow aSb\lambda A\lambda a$ $A \rightarrow Aa$ $m < n: S \rightarrow aSb\lambda b\lambda b$ $B \rightarrow Bb$ $m \geq n: S \rightarrow aSb\lambda A\lambda b$ $B \rightarrow Bb$	LLC (APD)	O marcador # torna a decisão determinística; o APD sabe exatamente quando parar de empilhar ou desempilhar para fazer a comparação.	Se o marcador # fosse removido, ela se tornaria a L_6 (LLC Não-Determinística).

G: ABC|A|B|C

A → Aa | λ

B → Bb | λ

Confusão Comum na Prova

C → Cc | λ

II. Linguagens Livres de Contexto (LLC) — Memória de Pilha (AP)

B. LLC Não-Determinísticas (LLC \ LAD)

#	Linguagem (L)	Descrição Matemática	Classificação	Critério Chave	Confusão Comum na Prova
6	Contagem Inversa (Palíndromos)	$L = \{ww^R \mid w \in \{a, b\}^*\}$ (Palíndromos) $G: S \rightarrow aS1bSb1\lambda$	LLC (APN)	Exige comparação simétrica reversa. O AP tem que adivinhar o centro w_{meio} (transição λ que testa se w é par/ímpar)	É LLC, mas não LAD, pois o APD não pode se dar ao luxo de ter transições compatíveis para adivinhar o centro.
7	Contagem livre de Ordem	$L = \{w \mid n_0(w) = n_1(w)\}$ (Balanceamento) $G: S \rightarrow aSb1bSa1Sba1baS1Sba1\lambda$	LLC (APN)	Exige contagem igual sem ordem fixa. O APN usa a pilha como um balanço (empilha 0s, desempilha 1s, ou vice-versa).	Não é Regular (LR) 13 14. Gramática GLC para esta linguagem é $P \rightarrow 0P1P \mid 1P0P \mid \lambda$ 15.
8	Contagem Inversa Simples	$L = \{0^m 1^n \mid m \geq n\}$	LLC (APN)	O AP precisa empilhar 0s e garantir que, no mínimo, todos os 1s sejam pareados. Reconhecida por APN, mas a falta de marcador torna difícil a construção de um APD 8 16.	Provada Não-LR 17 18.

$$L = \{ww^R \mid w \in \{a, b\}^*\} = \text{LLC}$$

$$S \rightarrow aS1bSb1\lambda$$

a b a | b a a

III. Fora de LLC (Não-LLC) —

#	Linguagem (L)	Descrição Matemática	Classificação	Critério Chave	Confusão Comum na Prova
9	Múltiplas Contagens Similares	$L = \{a^i b^j c^i \mid i \geq 0\}$	Não-LLC	Requer pareamento de três variáveis 19. Uma única pilha só pode parear duas contagens de forma independente. Exige Autômato com 2 Pilhas 20.	Confusão CRÍTICA. O Lema do Bombeamento para LLCs é usado para provar a falha 19.
10	Interseção de Contagens	$L = \{a^n b^n c^m\} \cap \{a^m b^n c^n\}$	Não-LLC	A interseção resulta em $\{a^n b^n c^n\}$ 21, que é a linguagem Não-LLC L_8 . LLCs não são fechadas sob interseção 22. LLCs não são 21.	Confusão com o fechamento: embora LRs sejam fechadas sob interseção 22, LLCs não são 21.
11	Duplicação de Sub-strings	$L = \{ww \mid w \in \{a, b\}^*\}$ (String duplicada) ou $w w^R w$	Não-LLC	Exige comparação posicional (o primeiro símbolo de w deve ser igual ao primeiro símbolo do segundo w). A pilha não consegue acessar o fundo da memória para essa comparação 21. não exige simetria, mas que c é a metade restante da com exatidão comparação complexa	Provada Não-LR 23 24. A falha no reconhecimento por AP se deve à necessidade de lembrar a primeira metade w após desempilhá-la (ou empilhamento duplo complexo).
12	Quadrados Perfeitos	$L = \{a^{n^2} \mid n \geq 0\}$	Não-LLC	Requer contagem complexa que exige muito mais que a memória empilhável do AP 25 26.	Provada Não-LR 25 26.
13	Linguagens Primas	$L = \{a^n \mid n \text{ é primo}\}$	Não-LLC	Exige verificação de propriedades aritméticas que excedem a capacidade de um AP 27.	Provada Não-LR 27.

Requer 2+ Pilhas (Múltiplos Contadores)

#	Linguagem (L)	Descrição Matemática	Classificação	Critério Chave	Confusão Comum na Prova
14	Contagem com Soma	$L = \{a^m b^n c^k \mid m = n+k\}$ $L = \{a^m b^n c^k \mid m = n + k\}$ duas pilhas $b(i) \quad \quad c(k)$ } a pilha de a apaga um de b .	LLC	O AP pode empilhar n de b e k de c (ou usar estados para gerenciar as duas contagens) e depois parear com m de a .	Parece Não-LLC, mas a soma pode ser gerenciada em uma única pilha. Provada Não-LR 28 29.
15	Duas contagens diferentes	$L = \{a^i b^j c^k \mid i = j \text{ ou } i = k\}$ $L = \{a^i b^j c^k \mid i = j \text{ ou } i = k\}$ $\sum a^i b^j c^k \} \cup \sum a^i b^j c^k \}$ i contagem LLC j contagem LLC pilha ELLC	LLC	É a união de duas LLCs: $\{a^i b^i c^*\}$ e $\{a^k b^k c^*\}$. LLCs são fechadas sob união 19.	Não é Determinística (LAD). O APN precisa adivinhar se $i = j$ ou $i = k$. 6: $S_1 \rightarrow a S b C \mid a \times \cup \quad 6: S_2 \rightarrow a S c \mid a B c \mid a \times$ $C \rightarrow C c \mid a \times \quad B \rightarrow B b \mid a \times$

$S' \rightarrow S_1 \mid S_2$ } cria um símbolo inicial novo
 $S_1 \rightarrow a S b C \lambda$
 $S_2 \rightarrow a S c \mid a B c \lambda$ } repete os regidos
 $C \rightarrow C c \lambda$
 $B \rightarrow B b \lambda$

são simples, intérpretes
 não existe ambiguidade

GR

(formato)

→ pode ser ambígua

vs

GLC

alguns níveis de terminais e não-Terminais

$S \rightarrow \lambda$
 $S \rightarrow a S b$
 $S \rightarrow S S$
 $S \rightarrow S a \mid a S$
 $S \rightarrow a$

$S \rightarrow \lambda$
 $A \rightarrow a B$ ou $A \rightarrow B a$
 $B \rightarrow a$

mas:

$A \rightarrow B$ não precisa (descritível)

$A \rightarrow B B X$ (+ de 1 variável)

$A \rightarrow A X$ (recursão - inválida)

loop
(não produz terminal)

GLC com FNC:

$S \rightarrow \lambda$
 $A \rightarrow AB$
 $B \rightarrow b$ mas
 $A \rightarrow Aa X$

2. Linguagens Reconhecidas por Autômatos de Pilha (AP)

Os Autômatos de Pilha (AP) – incluindo o determinístico (APD) e o não determinístico (APN) – reconhecem as Linguagens Livres de Contexto (LLC). As linguagens reconhecidas por um AP podem ser definidas por três critérios, dependendo se o critério de aceitação é o estado final, a pilha vazia, ou ambos.

Seja um AP M com estados finais i e alfabeto de entrada Σ , a relação de movimento é representada por \vdash^* (que significa zero ou mais passos de transição). A configuração instantânea de um AP é dada pela tripla $[e, w, p]$, onde e é o estado atual, w é o restante da palavra de entrada, e p é o conteúdo da pilha (topo → fundo)

a. Reconhecimento por Estado Final e Pilha Vazia (Padrão $L(M)$)

Esta definição exige que o AP termine em um estado final ($f \in F$) e com a pilha vazia (λ):

Para um Autômato de Pilha Determinístico (APD) $M = (E, \Sigma, \Gamma, \delta, i, F)$:

$$L(M) = \{w \in \Sigma^* | [i, w, \lambda] \stackrel{*}{\vdash} [f, \lambda, \lambda] \text{ para algum } f \in F\}$$

$i \rightarrow E \xrightarrow{\text{(em } D \text{ ou } t \text{ paus de transição)}} f$

Esta mesma notação $L(M)$ é usada para definir a Linguagem Reconhecida por um Autômato de Pilha Não Determinístico (APN) 13 ... :

$$L(M) = \{w \in \Sigma^* | [i, w, \lambda] \stackrel{*}{\vdash} [f, \lambda, \lambda] \text{ para algum } f \in F\}$$

Uma palavra w é dita ser aceita por M se ela satisfizer essa condição 11 12 .

b. Reconhecimento por Estado Final Apenas ($L_F(M)$)

Esta definição requer que o AP termine em um estado final ($f \in F$) após consumir toda a entrada, independentemente do conteúdo da pilha ($y \in \Gamma^*$):

Para um Autômato de Pilha Não Determinístico (APN) M :

$$L_F(M) = \{w \in \Sigma^* | [i, w, \lambda] \stackrel{*}{\vdash} [f, \lambda, y] \text{ para algum } f \in F, y \in \Gamma^*\}$$

c. Reconhecimento por Pilha Vazia Apenas ($L_V(M)$)

Esta definição requer que o AP termine com a pilha vazia (λ) após consumir toda a entrada, independentemente do estado final ($e \in E$):

Para um Autômato de Pilha Não Determinístico (APN) $M = (E, \Sigma, \Gamma, \delta, i)$:

$$L_V(M) = \{w \in \Sigma^* | [i, w, \lambda] \stackrel{*}{\vdash} [e, \lambda, \lambda] \text{ para algum } e \in E\}$$

16 17

É importante notar que, para linguagens reconhecidas por APNs, o reconhecimento por pilha vazia e estado final ($L(M)$) é equivalente ao reconhecimento por estado final ($L_F(M)$), e L pode ser reconhecida por pilha vazia se $L \cup \{\lambda\}$ puder ser reconhecida por pilha vazia (L_V) 18 .