

A **Hierarquia de Chomsky** é um modelo teórico que classifica as linguagens formais em diferentes tipos, baseado na **complexidade das regras gramaticais** que as definem. Ela foi proposta pelo linguista **Noam Chomsky** em 1956 e é composta por quatro níveis.

## Sumário

[Resumo dos tipos de Chomsky](#)

[Linguagem 3](#)

[Linguagem 2](#)

[Linguagem 1](#)

[Linguagem 0](#)

---

# Resumo da hierarquia de Chomsky

---

## 1. Linguagens Regulares (Tipo 3)

Características:

- São as mais simples da hierarquia.
- Podem ser reconhecidas por **autômatos finitos** (determinísticos ou não).
- Definidas por **expressões regulares**.
- As regras da gramática seguem o formato:
  - **Produção Direita:**  $A \rightarrow aB$  ou  $A \rightarrow a$
  - **Produção Esquerda:**  $A \rightarrow Ba$  ou  $A \rightarrow a$
  - $A$  e  $B$  são variáveis,  $a$  é um terminal.

Exemplo:

Imagine que queremos definir uma linguagem que aceita palavras formadas pela repetição de ' $a$ ' seguidas por ' $b$ ' (como " $aabb$ ", " $ab$ ", " $aaaabbb$ ", etc.).

**Gramática Regular:**

```
S → aS | bA
A → bA | ε
```

- $S \rightarrow aS$  permite repetir  $a$  várias vezes antes de trocar para  $b$ .
- $bA$  garante que ao trocar de  $a$  para  $b$ , apenas  $b$  pode continuar.
- $A \rightarrow bA \mid \epsilon$  permite que terminemos a sequência de  $b$ .

**Autômato Finito (Diagrama Simples):**

```
(estado inicial) → (q0) --a--> (q0) --b--> (q1) --b--> (q1) → (estado final)
```

**Palavras Aceitas:** "ab", "aabb", "aaabbb"

**Palavras Rejeitadas:** "ba", "aabba", "aaa"

## 2. Linguagens Livres de Contexto (Tipo 2)

Características:

- São mais complexas que as regulares.
- São reconhecidas por **autômatos de pilha**.
- Definidas por **gramáticas livres de contexto (GLC)**.
- As regras devem ter a forma:
  - $A \rightarrow \alpha$  (onde  $A$  é uma variável e  $\alpha$  pode ser qualquer sequência de variáveis e terminais).

Exemplo:

Considere uma linguagem que aceita **palíndromos** formados por 'a' e 'b' (como "aba", "abba", "aaabbbaa").

**Gramática Livre de Contexto:**

$$S \rightarrow aSa \mid bSb \mid a \mid b \mid \epsilon$$

- $S \rightarrow aSa$  e  $S \rightarrow bSb$  garantem que a palavra será espelhada.
- $S \rightarrow a \mid b \mid \epsilon$  define os casos base (strings unitárias ou vazias).

**Palavras Aceitas:** "aba", "abba", "aaabbbaa"

**Palavras Rejeitadas:** "aab", "abbba", "baaa"

## 3. Linguagens Sensíveis ao Contexto (Tipo 1)

Características:

- São mais poderosas que as livres de contexto.
- São reconhecidas por **máquinas de Turing lineares**.
- As regras podem ter **contexto**, ou seja, dependem do que está ao redor do símbolo sendo substituído.
- A gramática segue a forma:
  - $\alpha A \beta \rightarrow \alpha \gamma \beta$  (onde  $\alpha$ ,  $\beta$  e  $\gamma$  podem ser cadeias de símbolos e  $A$  é uma variável).

Exemplo:

Linguagem que aceita cadeias do formato " $a^n b^n c^n$ " (quantidade igual de 'a', 'b' e 'c', como "aaabbbccc").

## Gramática Sensível ao Contexto:

```
S → aSBC | abc
CB → BC
B → b
C → c
```

- $S \rightarrow aSBC$  adiciona uma nova instância de 'a', 'b', e 'c' de forma balanceada.
- $CB \rightarrow BC$  mantém 'b' e 'c' organizados corretamente.
- $B \rightarrow b$  e  $C \rightarrow c$  transformam os símbolos auxiliares em terminais.

**Palavras Aceitas:** "abc", "aabbcc", "aaabbbccc"

**Palavras Rejeitadas:** "aabb", "aaabbbccc", "abbc"

---

## 4. Linguagens Recursivamente Enumeráveis (Tipo 0)

Características:

- São as mais gerais e poderosas.
- Podem ser reconhecidas por uma **máquina de Turing**.
- As regras podem ser **quaisquer substituições** da forma:
  - $\alpha \rightarrow \beta$  (onde  $\alpha$  e  $\beta$  podem ser cadeias arbitrárias de variáveis e terminais).
- Esse tipo de linguagem inclui **problemas indecidíveis**, ou seja, que não podem ser resolvidos de forma automática para todos os casos.

Exemplo:

Linguagem que verifica se uma **expressão matemática** está correta ( $2+3*4$  é válido, mas  $+2*$  não).

**Gramática Recursivamente Enumerável:**

```
E → E+E | E*E | (E) | número
número → 0 | 1 | 2 | ... | 9
```

- Permite construir expressões matemáticas válidas recursivamente.
- Aceita qualquer combinação correta de operadores e operandos.

**Palavras Aceitas:** "3+4", "2\*(3+5)", "7+8\*2"

**Palavras Rejeitadas:** "++4", "3+\*5", "()\*4"

---

## Resumo da Hierarquia de Chomsky

Tipo	Nome	Modelo de Reconhecimento	Exemplo
------	------	--------------------------	---------

Tipo	Nome	Modelo de Reconhecimento	Exemplo
0	Recursivamente Enumeráveis	Máquina de Turing	Expressões matemáticas
1	Sensíveis ao Contexto	Máquina de Turing Linear	$a^n b^n c^n$
2	Livres de Contexto	Autômato de Pilha	Palíndromos ( <i>aba</i> , <i>abba</i> )
3	Regulares	Autômato Finito	$a^n b^n$

A **Hierarquia de Chomsky** organiza as linguagens formais de forma crescente em poder expressivo. Quanto maior o nível, mais difícil é analisar a linguagem, mas maior a capacidade de expressão. Linguagens de programação geralmente pertencem ao nível das livres de contexto, mas possuem restrições que podem exigir linguagens sensíveis ao contexto.

## Linguagens Regulares (Tipo 3) na Hierarquia de Chomsky

As **linguagens regulares** são o tipo mais simples na **Hierarquia de Chomsky** e desempenham um papel fundamental no desenvolvimento de **compiladores**, principalmente na **análise léxica**. Essas linguagens podem ser descritas por **expressões regulares** e reconhecidas por **autômatos finitos**.

### \*\* Características das Linguagens Regulares \*\*

- São definidas por **gramáticas regulares**, que possuem regras de produção restritas.
- Podem ser representadas por **expressões regulares**.
- São reconhecidas por **autômatos finitos** (determinísticos ou não).
- Possuem **memória limitada** → Não podem contar infinitamente, pois os autômatos finitos não possuem pilha ou fita de memória.

### \*\* Gramáticas Regulares \*\*

As gramáticas regulares seguem regras específicas para as suas produções:

- **Forma geral das produções:**
  - **Gramática regular à direita:**

```
A → aB
A → a
```

- **Gramática regular à esquerda:**

```
A → Ba
A → a
```

- **A** e **B** são variáveis (não terminais).
- **a** é um terminal.

**Importante:** Para ser uma **gramática regular**, todas as regras precisam seguir um único padrão (regular à direita ou regular à esquerda).

---

## \*\* Exemplo 1: Uma Linguagem de Palavras Simples\*\*

**Objetivo:** Criar uma linguagem que aceite palavras que começam com '**a**' e terminam com '**b**'.

### Gramática Regular:

```
S → aA  
A → b | aA
```

#### Palavras aceitas:

"ab", "aab", "aaab", "aaaab", etc.

#### Palavras rejeitadas:

"ba", "a", "abb", "bbb", etc.

#### Explicação:

- $S \rightarrow aA \rightarrow A$  palavra **sempre** começa com '**a**'.
- $A \rightarrow aA \rightarrow$  Pode repetir '**a**' quantas vezes quiser.
- $A \rightarrow b \rightarrow$  Quando aparece '**b**', a palavra termina.

## Autômato Finito Equivalente

Podemos desenhar um **autômato finito determinístico (AFD)** para reconhecer essa linguagem:

```
(q0) --a--> (q1) --a--> (q1) --b--> (q2) [final]
```

- **q0** → Estado inicial.
- **q1** → Aceita '**a**' e repete.
- **q2** → Aceita '**b**' e finaliza.

Se terminar em **q2**, a palavra é aceita.

---

## \*\* Exemplo 2: Identificadores de Variáveis\*\*

**Objetivo:** Criar uma linguagem que aceite identificadores válidos para uma linguagem de programação.

Suponha que um identificador deve começar com uma **letra** (a-z ou A-Z) e pode conter **letras ou números** depois disso.

### Gramática Regular:

```
S → L A  
A → L A | D A | ε
```

Onde:

- L representa uma **letra** (a-z, A-Z).
- D representa um **dígito** (0-9).

#### Palavras aceitas:

"x", "var1", "teste123", "A2B3"

#### Palavras rejeitadas:

"1teste", "123", "@var"

### Expressão Regular Equivalente

A mesma linguagem pode ser expressa por uma **expressão regular**:

```
[a-zA-Z][a-zA-Z0-9]*
```

- [a-zA-Z] → O primeiro caractere deve ser uma letra.
- [a-zA-Z0-9]\* → Os próximos podem ser letras ou números (zero ou mais vezes).

---

## \*\* Exemplo 3: Números Binários Pares\*\*

**Objetivo:** Criar uma linguagem que aceite números binários terminando em 0.

### Gramática Regular:

```
S → 0 | 1S0
```

#### Palavras aceitas:

"0", "10", "110", "1010", "10000"

#### Palavras rejeitadas:

"1", "11", "101", "111"

### Autômato Finito Equivalente

```
(q0) --0--> [q1] (final)
(q0) --1--> (q0) --0--> [q1] (final)
```

- O número pode ter vários 1s, mas **sempre** termina em 0.

---

## \*\* Limitações das Linguagens Regulares\*\*

Apesar de serem poderosas, as **linguagens regulares não conseguem expressar todas as linguagens possíveis**.

### Problema: Linguagens que exigem contagem

Exemplo: A linguagem "**a<sup>n</sup> b<sup>n</sup>**" (quantidade igual de 'a' e 'b', como "aabb", "aaabbb", etc.).

#### Por que não é regular?

- Para reconhecer "**a<sup>n</sup> b<sup>n</sup>**", seria necessário **contar** quantos 'a' e 'b' foram lidos.
- **Autômatos finitos não têm memória suficiente para contar.**
- Para isso, precisaríamos de um **autômato de pilha**, ou seja, uma **linguagem livre de contexto (Tipo 2 da Hierarquia de Chomsky)**.

---

## \*\* Aplicações das Linguagens Regulares\*\*

### Compiladores:

- A **análise léxica** de um compilador usa **expressões regulares** e **autômatos finitos** para reconhecer tokens (palavras-chave, identificadores, operadores, etc.).

### Validação de entrada:

- Verificar se um **e-mail** está no formato correto:

```
[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}
```

- Verificar se uma **placa de carro** segue um padrão (**AAA-1234**).

### Protocolos de Rede:

- Filtragem de pacotes e **expressões regulares** para detecção de padrões.

---

## Conclusão

As **linguagens regulares (Tipo 3 da Hierarquia de Chomsky)** são fundamentais na teoria da computação e em aplicações reais, como **compiladores, análise léxica e validação de dados**. Elas

podem ser descritas por **gramáticas regulares**, **expressões regulares** e **autômatos finitos**. No entanto, possuem **limitações**, pois não conseguem lidar com linguagens que exigem **memória para contagem**.

### Resumo:

Fáceis de implementar.

Úteis para análise léxica e validação de entrada.

Não conseguem lidar com linguagens mais complexas.

## Implementação em linguagem C

Aqui está um exemplo **em C** que implementa um reconhecedor de uma linguagem **regular (Tipo 3 da Hierarquia de Chomsky)**.

---

### \*\* Linguagem Regular do Tipo 3\*\*

Vamos definir a seguinte linguagem:

$L = \{ a^n b^n \mid n \geq 1 \}$

Ou seja, a linguagem aceita apenas cadeias onde **todos os as vêm antes dos bs** e o número de **as** deve ser igual ao número de **bs**.

### Exemplos de strings aceitas:

"ab", "aabb", "aaabbb", "aaaabbbb"

### Exemplos de strings rejeitadas:

X "abb", "ba", "aabbb", "aaaaabb"

A gramática **regular** para essa linguagem pode ser escrita assim:

$[S \rightarrow aS \mid \mid b]$

---

### \*\* Código em C\*\*

O código abaixo verifica se a string pertence à linguagem  $a^n b^n$ , seguindo a gramática regular do Tipo 3:

```
#include <stdio.h>
#include <string.h>

// Função para verificar se a string pertence à linguagem a^n b^n
int verifica_anbn(const char *str) {
    int count_a = 0, count_b = 0;
    int i = 0, estado = 0;

    // Percorre a string verificando a sequência correta
    while (str[i] != '\0') {
        if (estado == 0) {
            if (str[i] == 'a') {
```



```

        count_a++;
    } else if (str[i] == 'b') {
        estado = 1; // Troca para estado de 'b's
        count_b++;
    } else {
        return 0; // Se encontrar outro caractere, rejeita
    }
} else if (estado == 1) {
    if (str[i] == 'b') {
        count_b++;
    } else {
        return 0; // Se encontrar outro caractere, rejeita
    }
}
}
i++;
}

// A string é aceita se count_a == count_b e pelo menos um 'a' e um 'b'
existirem
return (count_a == count_b && count_a > 0);
}

int main() {
    // Testes com diferentes strings
    char *testes[] = { "ab", "aabb", "aaabbb", "aaaabbbb", "abb", "ba",
"aabbb", "aaaaabb", "bbb", "" };
    int num_testes = sizeof(testes) / sizeof(testes[0]);

    for (int i = 0; i < num_testes; i++) {
        if (verifica_anbn(testes[i]))
            printf("A string \"%s\" pertence à linguagem a^n b^n\n",
testes[i]);
        else
            printf("A string \"%s\" NÃO pertence à linguagem a^n b^n\n",
testes[i]);
    }

    return 0;
}

```

## **\*\* Exemplo de Saída\*\***

```

A string "ab" pertence à linguagem a^n b^n
A string "aabb" pertence à linguagem a^n b^n
A string "aaabbb" pertence à linguagem a^n b^n
A string "aaaabbbb" pertence à linguagem a^n b^n
A string "abb" NÃO pertence à linguagem a^n b^n
A string "ba" NÃO pertence à linguagem a^n b^n
A string "aabbb" NÃO pertence à linguagem a^n b^n

```

```
A string "aaaaabb" NÃO pertence à linguagem  $a^n b^n$   
A string "bbb" NÃO pertence à linguagem  $a^n b^n$   
A string "" NÃO pertence à linguagem  $a^n b^n$ 
```

---

## \*\* Explicação \*\*

O código verifica se a string contém somente **as** seguidos de **bs**.

Conta o número de **as** e **bs** e verifica se são iguais.

Rejeita strings que tenham **bs** antes dos **as** ou números diferentes de **as** e **bs**.

---

## \*\* Resumo \*\*

O programa **simula um autômato finito** que aceita a linguagem **regular  $a^n b^n$** .

**Segue a gramática regular do Tipo 3**  $S \rightarrow aS \mid \mid bS$ .

**Pode ser adaptado para outras linguagens regulares**, como  $(ab)^n$  ou  $(aa|bb)^n$ .

---

# Linguagens Livres de Contexto (Tipo 2) na Hierarquia de Chomsky

---

As **linguagens livres de contexto (LLC)** formam a **segunda camada da Hierarquia de Chomsky** e são fundamentais para o desenvolvimento de **compiladores**, pois descrevem a **sintaxe das linguagens de programação**.

---

## \*\* Características das Linguagens Livres de Contexto \*\*

- São definidas por gramáticas livres de contexto (GLC).
  - Podem ser reconhecidas por autômatos de pilha (PDA - Pushdown Automata).
  - Podem lidar com estruturas recursivas, como parênteses aninhados ou expressões matemáticas.
  - Possuem mais poder que as linguagens regulares (Tipo 3), pois podem contar informações (graças à pilha do autômato).
- 

## \*\* Gramáticas Livres de Contexto (GLC) \*\*

Uma gramática livre de contexto é definida por **regras de produção** da forma:

```
A → α
```

Onde:

- **A** é um **não-terminal** (ou variável).
-

- $\alpha$  é uma sequência de **terminais e/ou não-terminais**.

**Importante:** O lado esquerdo da produção **sempre tem um único não-terminal**.

### \*\* Exemplo 1: Linguagem de Parênteses Bem Formados \*\*

**Objetivo:** Criar uma linguagem que aceita expressões com **parênteses aninhados corretamente**.

## Gramática Livre de Contexto

$$S \rightarrow (S)S \mid \varepsilon$$

**Palavras aceitas:**

"()", "(() )", "()( )", "(( ) ( ) )", "" (vazio).

**Palavras rejeitadas:**

)(","())(","((","")())".

**Explicação:**

- $S \rightarrow (S)S$  permite criar expressões aninhadas.
- $\epsilon$  (vazio) indica que a recursão pode parar.

## Autômato de Pilha Equivalente

- Para cada '(', **empilha** um marcador.
- Para cada ')', **desempilha** um marcador.
- Se a pilha estiver vazia no final, a string é válida.

Se terminar com a pilha vazia → palavra aceita.

Se houver algo na pilha → palavra rejeitada.

## \*\* Exemplo 2: Linguagem de Palavras $a^n b^n$ \*\*

**Objetivo:** Criar uma linguagem que aceita palavras com um número **igual de 'a' e 'b'**.

## Gramática Livre de Contexto

$$S \rightarrow aSb \mid \epsilon$$

**Palavras aceitas:**

"ab", "aabb", "aaabbb", "aaaabbbb".

**Palavras rejeitadas:**

"aab", "abb", "bba", "aaabb".

### Explicação:

- $S \rightarrow aSb$  adiciona 'a' no início e 'b' no final.
- $\epsilon$  permite que a recursão pare.

### Autômato de Pilha Equivalente

- Para cada 'a', **empilha** um marcador.
- Para cada 'b', **desempilha** um marcador.
- Se a pilha ficar **vazia no final**, a palavra é aceita.

Se a pilha estiver vazia  $\rightarrow$  palavra aceita.

Se sobrarem 'a's ou 'b's  $\rightarrow$  palavra rejeitada.

## \*\* Exemplo 3: Expressões Matemáticas Simples\*\*

**Objetivo:** Criar uma linguagem que reconhece expressões matemáticas simples, como "1+2", "3\*(4+5)".

### Gramática Livre de Contexto

```
E  $\rightarrow$  E + T | E - T | T  
T  $\rightarrow$  T * F | T / F | F  
F  $\rightarrow$  (E) | num
```

#### Palavras aceitas:

"1+2", "3\*(4+5)", "8-6/2".

#### Palavras rejeitadas:

"+1", "3\*/2", "())".

### Explicação:

- E representa **expressões**.
- T representa **termos**.
- F representa **fatores** (números ou expressões entre parênteses).

### Árvore Sintática para 3\*(4+5)

```
      E  
     /\ \  
    T * F  
    |  |  
    3  (E)  
       /\ \  
      E + T  
      |  |  
      4  5
```

**Importância:** Esta gramática permite que o compilador construa **árvores sintáticas** para interpretar expressões matemáticas corretamente.

## \*\* Diferença entre Linguagens Regulares e Linguagens Livres de Contexto\*\*

Característica	Linguagens Regulares (Tipo 3)	Linguagens Livres de Contexto (Tipo 2)
<b>Expressividade</b>	Simples (não podem contar)	Mais complexas (permitem contagem)
<b>Reconhecimento</b>	Autômato Finito (DFA/NFA)	Autômato de Pilha (PDA)
<b>Recursão</b>	Não permite	Permite
<b>Exemplo</b>	"a*b"	"a^n b^n"

## \*\* Aplicações das Linguagens Livres de Contexto\*\*

### Compiladores:

- A **análise sintática** em compiladores usa gramáticas livres de contexto para estruturar programas.

### Linguagens de Programação:

- A estrutura de expressões matemáticas, laços e comandos é descrita por gramáticas livres de contexto.

### Análise de Sentenças na Linguística:

- Em linguística computacional, GLCs são usadas para analisar a gramática de frases.

### Verificação de Parênteses e HTML:

- Verificação de expressões matemáticas e estruturas aninhadas como XML/HTML.

## \*\* Conclusão\*\*

As **linguagens livres de contexto** (Tipo 2 da Hierarquia de Chomsky) são **mais poderosas** que as linguagens regulares, pois podem representar **estruturas recursivas e contagens**. Elas são essenciais para **compiladores, linguagens de programação e linguística computacional**.

### Resumo:

Podem expressar **estruturas aninhadas**.

São processadas por **autômatos de pilha**.

Fundamentais para **análise sintática** em compiladores.

**Não conseguem lidar com certas linguagens mais complexas**, como 'a^n b^n c^n' (requer gramáticas sensíveis ao contexto - Tipo 1).

## \*\* Implementação de um Reconhecedor para uma Linguagem Livre de Contexto (Tipo 2) em C\*\*

A **Gramática do Tipo 2** (Linguagens Livres de Contexto) na **Hierarquia de Chomsky** pode ser reconhecida por um **Autômato de Pilha (PDA)**.

---

### \*\* Escolha da Linguagem\*\*

Vamos implementar um reconhecedor para a linguagem:

$L = \{ a^n b^n \mid n \geq 1 \}$

Isso significa que a string deve conter um número **igual** de **as** e **bs**, **mas sempre com os as antes dos bs**.

#### Exemplos de strings aceitas:

- **ab**
- **aabb**
- **aaabbb**
- **aaaabbbb**

#### Exemplos de strings rejeitadas:

- **abb** (falta um **a**)
  - **ba** (ordem errada)
  - **aab** (falta um **b**)
- 

### \*\* Código em C\*\*

O programa utiliza **uma pilha** para simular um **Autômato de Pilha (PDA)**. Cada **a** é empilhado e cada **b** desempilha um **a**. No final, a pilha deve estar **vazia** para a string ser aceita.

```
#include <stdio.h>
#include <string.h>

#define MAX 100

// Estrutura da Pilha
typedef struct {
    char dados[MAX];
    int topo;
} Pilha;

// Inicializa a pilha
void inicializar_pilha(Pilha *p) {
    p->topo = -1;
}

// Verifica se a pilha está vazia
int pilha_vazia(Pilha *p) {
```

```

    return (p->topo == -1);
}

// Adiciona um elemento na pilha
void empilhar(Pilha *p, char c) {
    if (p->topo < MAX - 1) {
        p->dados[++p->topo] = c;
    }
}

// Remove um elemento da pilha
char desempilhar(Pilha *p) {
    if (!pilha_vazia(p)) {
        return p->dados[p->topo--];
    }
    return '\0'; // Retorna vazio se a pilha estiver vazia
}

// Função que verifica se a string pertence à linguagem a^n b^n
int reconhece_anbn(const char *str) {
    Pilha pilha;
    inicializar_pilha(&pilha);
    int i, len = strlen(str);

    // Primeira fase: empilhar todos os 'a's
    for (i = 0; i < len; i++) {
        if (str[i] == 'a') {
            empilhar(&pilha, 'a'); // Armazena os 'a's na pilha
        } else if (str[i] == 'b') {
            if (pilha_vazia(&pilha)) {
                return 0; // Se não há 'a' para desempilhar, a string é
                inválida
            }
            desempilhar(&pilha); // Remove um 'a' para cada 'b'
        } else {
            return 0; // Caracter inválido na entrada
        }
    }

    // Se a pilha estiver vazia no final, a string pertence à linguagem
    return pilha_vazia(&pilha);
}

int main() {
    // Testes
    char *testes[] = { "ab", "aabb", "aaabbb", "aaaabbbb", "abb", "ba", "aab",
    "aaaaabb", "" };
    int num_testes = sizeof(testes) / sizeof(testes[0]);

    for (int i = 0; i < num_testes; i++) {
        if (reconhece_anbn(testes[i]))
            printf("A string \"%s\" pertence à linguagem a^n b^n\n",
            testes[i]);
    }
}

```

```

        else
            printf("A string \"%s\" NÃO pertence à linguagem a^n b^n\n",
testes[i]);
        }

        return 0;
    }

```

**\*\* Como funciona?\*\***

**Empilha os **a**s encontrados.**

**Para cada **b** encontrado, desempilha um **a**.**

**Se a pilha estiver vazia no final, a string pertence à linguagem.**

**\*\* Saída esperada\*\***

```

A string "ab" pertence à linguagem a^n b^n
A string "aabb" pertence à linguagem a^n b^n
A string "aaabbb" pertence à linguagem a^n b^n
A string "aaaabbbb" pertence à linguagem a^n b^n
A string "abb" NÃO pertence à linguagem a^n b^n
A string "ba" NÃO pertence à linguagem a^n b^n
A string "aab" NÃO pertence à linguagem a^n b^n
A string "aaaaabb" NÃO pertence à linguagem a^n b^n
A string "" NÃO pertence à linguagem a^n b^n

```

**\*\* Explicação\*\***

**Autômato de Pilha:** Utiliza uma pilha para armazenar os **a**s e os desempilha ao encontrar **b**s.

**Ordem Importante:** Os **b**s nunca podem vir antes dos **a**s.

**Conta igual de **a**s e **b**s:** Se sobrar **a**s na pilha no final, a string é rejeitada.

**\*\* Resumo\*\***

**Este é um exemplo de gramática do Tipo 2 da Hierarquia de Chomsky.**

**Utiliza um Autômato de Pilha (PDA)** para reconhecer a linguagem.

**Simples, eficiente e fácil de expandir** para outras linguagens livres de contexto.

## Gramáticas Sensíveis ao Contexto (Tipo 1) – Hierarquia de Chomsky

As **linguagens sensíveis ao contexto** (LSC) formam o **Tipo 1** da **Hierarquia de Chomsky** e são mais poderosas que as **linguagens livres de contexto**.



Elas são usadas para **linguagens formais que precisam de regras mais rigorosas**, como certas estruturas da **linguagem natural** e algumas partes de **compiladores**.

---

## \*\* Características das Linguagens Sensíveis ao Contexto \*\*

- **Definidas por gramáticas sensíveis ao contexto (GSC).**
  - **Podem ser reconhecidas por máquinas de Turing lineares** (LBA – *Linear Bounded Automaton*).
  - **As regras de produção dependem do contexto**, ou seja, um símbolo pode ser substituído **somente se estiver dentro de um contexto específico**.
  - **Podem representar linguagens que exigem contagens múltiplas**, como  $a^n b^n c^n$ .
- 

## \*\* Regras das Gramáticas Sensíveis ao Contexto \*\*

As regras de produção têm a seguinte forma:

$$\alpha A \beta \rightarrow \alpha \gamma \beta$$

Onde:

- $A$  é um **não-terminal**.
- $\alpha$  e  $\beta$  são cadeias de **terminais e não-terminais** (o contexto).
- $\gamma$  é uma cadeia de **terminais e/ou não-terminais**, com **tamanho igual ou maior que  $A$** .

**Importante:**

O lado direito **não pode ser menor** que o lado esquerdo, garantindo que a cadeia nunca diminua.

---

## \*\* Exemplo 1: Linguagem $a^n b^n c^n$ \*\*

**Objetivo:** Criar uma linguagem que aceita cadeias com **o mesmo número de 'a', 'b' e 'c'**, como:

"abc", "aabbcc", "aaabbbccc".

"aabbcc", "abbcc", "aaabbbcc".

---

## Gramática Sensível ao Contexto

```
S → aSBC | abc
CB → BC
B → b
C → c
```

**Explicação:**

- $S \rightarrow aSBC$  insere 'a' no início e 'BC' no final, garantindo o padrão  $a^n B^n C^n$ .
-

- $CB \rightarrow BC$  reorganiza a cadeia para 'b's ficarem antes dos 'c's.
- $B \rightarrow b$  e  $C \rightarrow c$  transformam os não-terminais 'B' e 'C' em seus respectivos símbolos terminais.

### Derivação para "aaabbbccc"

1.  $S \rightarrow aSBC \rightarrow "aSBC"$
2.  $aSBC \rightarrow aaSBCCB \rightarrow "aaSBCCB"$
3.  $aaSBCCB \rightarrow aaaBCCBBC$
4. **Reorganização:**
  - $CB \rightarrow BC \rightarrow "aaaBBBCCC"$
5. **Transformação:**
  - $B \rightarrow b, C \rightarrow c \rightarrow "aaabbbccc"$

### Por que isso NÃO é possível com gramáticas livres de contexto (GLC - Tipo 2)?

Porque **GLCs não conseguem controlar múltiplas contagens simultaneamente**. A linguagem  $a^n b^n c^n$  exige que as contagens de 'a', 'b' e 'c' sejam **iguais**, o que não pode ser feito apenas com uma pilha (como nos autômatos de pilha).

## \*\* Exemplo 2: Verificação de Ordem de Palavras \*\*

**Objetivo:** Criar uma linguagem que aceita frases onde um adjetivo só pode ser seguido por um substantivo.

"grande carro"

"pequeno gato"

"carro grande"

"gato pequeno"

### Gramática Sensível ao Contexto

```
S → ADJ N
ADJ → grande | pequeno
N → carro | gato
```

### Por que essa gramática é sensível ao contexto?

Porque a regra  $S \rightarrow ADJ N$  garante que **um adjetivo só pode aparecer antes de um substantivo**. Se fosse livre de contexto, permitiria qualquer ordem.

## \*\* Diferença entre os Tipos 2 e 1 (GLC vs. GSC) \*\*

Característica	Linguagens Livres de Contexto (Tipo 2)	Linguagens Sensíveis ao Contexto (Tipo 1)
----------------	---	--

Característica	Linguagens Livres de Contexto (Tipo 2)	Linguagens Sensíveis ao Contexto (Tipo 1)
<b>Forma das Regras</b>	$A \rightarrow \alpha$ (um único não-terminal vira algo)	$\alpha A \beta \rightarrow \alpha \gamma \beta$ (depende do contexto)
<b>Reconhecedor</b>	Autômato de Pilha (PDA)	Máquina de Turing Linear (LBA)
<b>Capacidade</b>	Contagem simples (ex: $a^n b^n$ )	Contagem múltipla (ex: $a^n b^n c^n$ )
<b>Exemplo</b>	" $a^n b^n$ " (n cópias de 'a' e 'b')	" $a^n b^n c^n$ " (n cópias de 'a', 'b' e 'c')

## \*\* Aplicações das Linguagens Sensíveis ao Contexto \*\*

### Processamento de Linguagem Natural (PLN):

- Algumas regras gramaticais humanas precisam de contexto (ex: gênero e número em frases).

### Verificação de Código Fonte:

- Algumas restrições de sintaxe e escopo em linguagens de programação podem ser descritas com gramáticas sensíveis ao contexto.

### Reconhecimento de Padrões em DNA:

- Algumas estruturas complexas de DNA seguem padrões que podem ser modelados com GSC.

## \*\* Conclusão \*\*

As **linguagens sensíveis ao contexto** (Tipo 1) são **mais poderosas** que as linguagens livres de contexto (Tipo 2), pois podem descrever **estruturas que exigem múltiplas contagens simultâneas**.

### Resumo:

**Expressam linguagens que GLCs não conseguem**, como  $a^n b^n c^n$ .

**Usam regras de produção dependentes do contexto.**

**Reconhecidas por Máquinas de Turing Lineares (LBA).**

**Menos eficientes que GLCs**, tornando sua aplicação prática mais difícil.

# Gramáticas Recursivamente Enumeráveis (Tipo 0) – Hierarquia de Chomsky

As **Linguagens Recursivamente Enumeráveis** formam o **Tipo 0** da **Hierarquia de Chomsky** e são o tipo **mais geral e poderoso** de linguagens formais.

Elas correspondem a qualquer **linguagem que pode ser reconhecida por uma Máquina de Turing** e, portanto, representam **tudo o que é computável** dentro do modelo tradicional de computação.

## **\*\* Linguagem do Tipo 1 (Sensível ao Contexto)\*\***

A linguagem escolhida é:

$L = \{ a^n b^m c^m d^n \mid n, m \geq 1 \}$

Essa linguagem exige que:

1. O número de **as** seja **igual** ao número de **ds**.
2. O número de **bs** seja **igual** ao número de **cs**.
3. A ordem seja **sempre**: **a...b...c...d**.

Essa é uma linguagem **sensível ao contexto** porque **o número de as afeta o número de ds**, e **o número de bs afeta o número de cs**.

---

## **\*\* Código em C\*\***

```
#include <stdio.h>
#include <string.h>

// Função para verificar se a string pertence à linguagem a^n b^m c^m d^n
int verifica_linguagem(const char *str) {
    int count_a = 0, count_b = 0, count_c = 0, count_d = 0;
    int i = 0, estado = 0;

    // Percorre a string verificando os caracteres e contando
    while (str[i] != '\0') {
        if (estado == 0) { // Contando 'a's
            if (str[i] == 'a') count_a++;
            else if (str[i] == 'b') { estado = 1; count_b++; }
            else return 0; // Erro de sequência ou caractere inválido
        }
        else if (estado == 1) { // Contando 'b's
            if (str[i] == 'b') count_b++;
            else if (str[i] == 'c') { estado = 2; count_c++; }
            else return 0;
        }
        else if (estado == 2) { // Contando 'c's
            if (str[i] == 'c') count_c++;
            else if (str[i] == 'd') { estado = 3; count_d++; }
            else return 0;
        }
        else if (estado == 3) { // Contando 'd's
            if (str[i] == 'd') count_d++;
            else return 0; // Qualquer outro caractere é inválido
        }
        i++;
    }
}
```

```

// Verifica se os números seguem a regra n = d e b = c
return (count_a == count_d && count_b == count_c && count_a >= 1 && count_b
>= 1);
}

int main() {
    // Casos de teste
    char *testes[] = { "abbcdd", "aabbccdd", "aaabbccdd", "abc", "aabbccddd",
"aaaabbbbccccddddd", "aabbccdd", "aaaa" };
    int num_testes = sizeof(testes) / sizeof(testes[0]);

    for (int i = 0; i < num_testes; i++) {
        if (verifica_linguagem(testes[i]))
            printf("A string \"%s\" pertence à linguagem a^n b^m c^m d^n\n",
testes[i]);
        else
            printf("A string \"%s\" NÃO pertence à linguagem a^n b^m c^m
d^n\n", testes[i]);
    }

    return 0;
}

```

## \*\* Exemplo de Saída \*\*

```

A string "abbcdd" pertence à linguagem a^n b^m c^m d^n
A string "aabbccdd" pertence à linguagem a^n b^m c^m d^n
A string "aaabbccdd" pertence à linguagem a^n b^m c^m d^n
A string "abc" NÃO pertence à linguagem a^n b^m c^m d^n
A string "aabbccddd" NÃO pertence à linguagem a^n b^m c^m d^n
A string "aaaabbbbccccddddd" pertence à linguagem a^n b^m c^m d^n
A string "aabbccdd" NÃO pertence à linguagem a^n b^m c^m d^n
A string "aaaa" NÃO pertence à linguagem a^n b^m c^m d^n

```

## \*\* Explicação \*\*

**Conta os caracteres a, b, c e d.**

**Verifica se estão na ordem correta.**

**Garante que as e ds tenham o mesmo número.**

**Garante que bs e cs tenham o mesmo número.**

**Rejeita strings que não seguem a regra  $a^n b^m c^m d^n$ .**

## \*\* Resumo \*\*

Este código implementa **uma linguagem sensível ao contexto** do **Tipo 1** da Hierarquia de Chomsky.

**Verifica a validade da entrada** com base nas regras da gramática.

**Usa uma abordagem eficiente** de contagem de caracteres e estados.

---

## \*\* Características das Linguagens Tipo 0 \*\*

- **Definidas por Gramáticas Recursivamente Enumeráveis (GRE).**
  - **Podem ser reconhecidas por uma Máquina de Turing (MT)**, mas nem sempre em tempo finito.
  - **Regras de produção são totalmente livres**, podendo transformar qualquer sequência de símbolos em outra.
  - **Podem descrever qualquer linguagem formal computável, incluindo problemas indecidíveis.**
- 

## \*\* Regras das Gramáticas Recursivamente Enumeráveis \*\*

As regras de produção são da forma:

$$\alpha \rightarrow \beta$$

Onde:

- $\alpha$  e  $\beta$  são **cadeias arbitrárias de terminais e não-terminais**, incluindo a possibilidade de  $\alpha$  ser qualquer sequência de símbolos.
- **Não há restrição de tamanho**, ou seja,  $\beta$  pode ser maior, menor ou igual a  $\alpha$ .

**Diferente dos outros tipos de gramática, não há garantia de que a derivação termine**, pois a Máquina de Turing pode **entrar em um loop infinito**.

---

## \*\* Exemplo 1: Reconhecendo o Palíndromo Geral \*\*

**Objetivo:** Criar uma gramática que reconhece qualquer palavra palíndroma, como:

"abba", "racecar", "madam".

"abc", "hello", "xyz".

---

### Gramática Recursivamente Enumerável

$$S \rightarrow aSa \mid bSb \mid cSc \mid \epsilon$$

**Explicação:**

- $S \rightarrow aSa$  adiciona 'a' no início e no fim.
  - $S \rightarrow bSb$  adiciona 'b' no início e no fim.
-

- $S \rightarrow cSc$  adiciona 'c' no início e no fim.
- $S \rightarrow \epsilon$  indica que uma string vazia também é válida (fim da recursão).

### Derivação para "racecar"

1.  $S \rightarrow rSr$
2.  $rSr \rightarrow raSar$
3.  $raSar \rightarrow racScar$
4.  $racScar \rightarrow raceEcar$
5.  $raceEcar \rightarrow racecar$

### Por que essa gramática precisa ser Tipo 0?

Porque não há **limite fixo** para quantas repetições podem ocorrer, e a recursão pode exigir processamento que só uma **Máquina de Turing** pode garantir.

## \*\* Exemplo 2: Linguagem $\{ a^n b^m c^m d^n \mid n, m \geq 1 \}^{**}$

**Objetivo:** Criar uma linguagem onde:

- A quantidade de 'a's é igual à de 'd's.
- A quantidade de 'b's é igual à de 'c's.
- 'n' e 'm' podem ser diferentes, mas devem ser **iguais dentro de seus pares**.

"abbccd", "aaabbbccddd".

"abbddd", "aabbccdd".

## Gramática Recursivamente Enumerável

```
S → aSd | T
T → bTc | bc
```

**Explicação:**

- $S \rightarrow aSd$  garante que **cada 'a' gerado tem um 'd' correspondente**.
- $T \rightarrow bTc$  garante que **cada 'b' gerado tem um 'c' correspondente**.
- $T \rightarrow bc$  fecha a recursão ao criar pares 'bc'.

### Derivação para "aabbccdd"

1.  $S \rightarrow aSd \rightarrow "aSd"$
2.  $aSd \rightarrow aaSdd \rightarrow "aaSdd"$
3.  $aaSdd \rightarrow aaTdd \rightarrow "aaTdd"$
4.  $aaTdd \rightarrow aabTcdd \rightarrow "aabTcdd"$
5.  $aabTcdd \rightarrow aabbccdd$

## Por que essa gramática é Tipo 0?

Porque é necessário um controle de múltiplas dependências (a/d e b/c), algo que apenas uma **Máquina de Turing** pode garantir em todos os casos.

## \*\* Diferença entre os Tipos 1 e 0 (GSC vs. GRE)\*\*

Característica	Linguagens Sensíveis ao Contexto (Tipo 1)	Linguagens Recursivamente Enumeráveis (Tipo 0)
Forma das Regras	$\alpha A \beta \rightarrow \alpha \gamma \beta$ (mantém contexto)	$\alpha \rightarrow \beta$ (qualquer transformação)
Reconhecedor	Máquina de Turing Linear (LBA)	Máquina de Turing Universal
Capacidade	Contagem e contexto limitado	Qualquer problema computável
Exemplo	$a^n b^n c^n$ (n cópias de cada)	Palíndromos gerais, problemas matemáticos complexos

## \*\* Aplicações das Linguagens Recursivamente Enumeráveis\*\*

### Teoria da Computação

- Muitas **linguagens formais e problemas matemáticos** caem na classe das **linguagens recursivamente enumeráveis**.

### Resolução de Problemas Complexos

- Qualquer **problema que pode ser resolvido por uma Máquina de Turing** pertence a essa classe.

### Linguagens de Programação (Análise Geral)

- Enquanto compiladores trabalham geralmente com **Gramáticas Livres de Contexto (GLC - Tipo 2)**, a **análise mais geral de linguagens** pode envolver gramáticas **Tipo 0**.

## \*\* Relação com Problemas Indecidíveis\*\*

A classe **Tipo 0** inclui **linguagens indecidíveis**, ou seja, problemas que **não podem ser resolvidos por qualquer algoritmo finito**.

Exemplo: **Problema da Parada**

- Dado um programa **P** e uma entrada **x**, existe um algoritmo que decide se **P(x)** para ou entra em um loop infinito?
- Resposta:** Não, porque **não existe Máquina de Turing que resolva isso para todos os casos**.

**Isso significa que há linguagens Tipo 0 que não podem ser reconhecidas em tempo finito!**



## **\*\* Conclusão \*\***

As **linguagens recursivamente enumeráveis** (Tipo 0) são as mais poderosas da Hierarquia de Chomsky.

**Permitem qualquer transformação em suas regras de produção.**

**Podem representar qualquer problema computável.**

**Reconhecidas por Máquinas de Turing Universais.**

**Podem ser indecidíveis, ou seja, alguns problemas não podem ser resolvidos em tempo finito.**

### **Resumo Final da Hierarquia de Chomsky**

Tipo	Nome	Reconhecedor	Exemplo
0	Recursivamente Enumeráveis	Máquina de Turing Universal	Palíndromos gerais, problemas indecidíveis
1	Sensíveis ao Contexto	Máquina de Turing Linear (LBA)	$a^n b^n c^n$
2	Livres de Contexto	Autômato de Pilha	Expressões matemáticas
3	Regulares	Autômato Finito	Palavras terminando em "ab"

Aqui está um exemplo em **C** que implementa um reconhecedor para uma linguagem do **Tipo 0** da **Hierarquia de Chomsky**. A linguagem escolhida é:

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

Isso significa que a linguagem aceita **qualquer palavra composta por a e b, seguida de seu espelhamento (reverso)**.

Exemplos de palavras aceitas:

- abbaabba
- aabbbbaa
- bababbab

## **\*\* Estratégia da Implementação \*\***

O programa verifica se a string de entrada pertence à linguagem  $ww^R$  seguindo estes passos:

**Divide a string ao meio.**

**Compara a segunda metade com o inverso da primeira metade.**

**Se forem idênticas, a string pertence à linguagem.**

## **\*\* Código em C \*\***

```
#include <stdio.h>
#include <string.h>

// Função que verifica se uma string pertence à linguagem ww^R
```

```

int verifica_wwR(const char *str) {
    int len = strlen(str);

    // Se o comprimento for ímpar, não pode ser da forma ww^R
    if (len % 2 != 0)
        return 0;

    int mid = len / 2; // Ponto central da string

    // Comparação da primeira metade com o reverso da segunda metade
    for (int i = 0; i < mid; i++) {
        if (str[i] != str[len - i - 1]) {
            return 0; // Se houver uma diferença, a string não pertence à
linguagem
        }
    }

    return 1; // A string pertence à linguagem
}

int main() {
    // Testes para validar a implementação
    char *testes[] = {
        "abbaabba", // Aceito (w = abba, w^R = abba)
        "aabbbaa",  // Aceito (w = aabb, w^R = baa)
        "bababbab", // Aceito (w = babab, w^R = babab)
        "abcba",    // Rejeitado (ímpar)
        "aabb",     // Rejeitado (não é ww^R)
        "bbaa",     // Rejeitado (não é ww^R)
        "abababab", // Aceito (w = abab, w^R = abab)
        ""          // Rejeitado (vazio)
    };

    int num_testes = sizeof(testes) / sizeof(testes[0]);

    for (int i = 0; i < num_testes; i++) {
        if (verifica_wwR(testes[i]))
            printf("A string \"%s\" pertence à linguagem ww^R\n", testes[i]);
        else
            printf("A string \"%s\" NÃO pertence à linguagem ww^R\n",
testes[i]);
    }

    return 0;
}

```

---

**\*\* Saída esperada\*\***

A string "abbaabba" pertence à linguagem  $ww^R$   
A string "aabbbaa" pertence à linguagem  $ww^R$   
A string "bababbab" pertence à linguagem  $ww^R$   
A string "abcba" NÃO pertence à linguagem  $ww^R$   
A string "aabb" NÃO pertence à linguagem  $ww^R$   
A string "bbaa" NÃO pertence à linguagem  $ww^R$   
A string "abababab" pertence à linguagem  $ww^R$   
A string "" NÃO pertence à linguagem  $ww^R$

---

**\*\* Explicação \*\***

**Apenas strings da forma  $ww^R$  são aceitas.**

**Usamos a técnica de espelhamento para validar a linguagem.**

**Não há restrições na gramática, sendo Tipo 0 da Hierarquia de Chomsky.**