Tutorial Breve sobre

Teoria da Computação

**em Português (PT-PT)**

**Uma imagem com texto, quadro preto, escrita à mão, Tipo de letra

Os conteúdos gerados por IA poderão estar incorretos.**

**10 Tópicos** Principais

[Uma imagem com símbolo, Tipo de letra, Gráficos, captura de ecrã

Os conteúdos gerados por IA poderão estar incorretos.](https://creativecommons.org/licenses/by-nc/4.0/)Luís Simões da Cunha

Uma imagem com texto, captura de ecrã, Tipo de letra, número

Os conteúdos gerados por IA poderão estar incorretos.

Índice

[🎯 **Os 10 Tópicos Essenciais em Teoria da Computação** 7](#_Toc195921440)

[🔍 1. **Linguagens Regulares e Autómatos Finitos** 9](#_Toc195921441)

[🧠 Visão Geral 9](#_Toc195921442)

[🧩 1.1 Definições formais de DFA e NFA 9](#_Toc195921443)

[📘 DFA – Autómato Finito Determinista 9](#_Toc195921444)

[🎲 NFA – Autómato Finito Não Determinista 9](#_Toc195921445)

[🔁 1.2 Equivalência entre DFA, NFA e Expressões Regulares 10](#_Toc195921446)

[🧼 1.3 Minimização de Autómatos 10](#_Toc195921447)

[🧮 O problema: 10](#_Toc195921448)

[✂️ A solução: 10](#_Toc195921449)

[🧠 1.4 Importância prática: Análise Léxica de Compiladores 10](#_Toc195921450)

[🎯 Conclusão e Ligações Pedagógicas 11](#_Toc195921451)

[💡 Atividades sugeridas: 11](#_Toc195921452)

[🔤 2. **Expressões Regulares e Gramáticas Regulares** 12](#_Toc195921453)

[💬 O que são Expressões Regulares? 12](#_Toc195921454)

[📏 2.1 Potência expressiva das REs 12](#_Toc195921455)

[🎨 Operadores fundamentais: 12](#_Toc195921456)

[🔄 2.2 Correspondência com Autómatos 12](#_Toc195921457)

[🎓 Implicações práticas: 13](#_Toc195921458)

[🔧 2.3 Algoritmos de Conversão entre REs e AFDs 13](#_Toc195921459)

[🛠️ RE → NFA-ε (construção direta) 13](#_Toc195921460)

[📘 NFA → DFA (algoritmo de subconjuntos) 13](#_Toc195921461)

[♻️ DFA → RE (eliminação de estados) 13](#_Toc195921462)

[🌍 2.4 Papel crucial em motores de busca e validação de padrões 13](#_Toc195921463)

[💡 Exemplos úteis do mundo real: 14](#_Toc195921464)

[📌 Resumo visual 14](#_Toc195921465)

[🧪 Desafios para ti: 14](#_Toc195921466)

[🧾 3. **Gramáticas Livre de Contexto (CFG) e Análise Sintática** 15](#_Toc195921467)

[📘 3.1 O que é uma CFG? 15](#_Toc195921468)

[🌳 3.2 Derivações, Árvores Sintáticas e Ambiguidade 15](#_Toc195921469)

[🔁 Derivações 15](#_Toc195921470)

[🌲 Árvores Sintáticas (Parse Trees) 15](#_Toc195921471)

[⚠️ Ambiguidade 16](#_Toc195921472)

[📏 3.3 Normalizações (Chomsky e Greibach) 16](#_Toc195921473)

[🔹 Forma Normal de Chomsky (CNF) 16](#_Toc195921474)

[🔹 Forma Normal de Greibach (GNF) 16](#_Toc195921475)

[🧠 3.4 Importância nas Linguagens de Programação 16](#_Toc195921476)

[🔍 3.5 Técnicas de Parsing (LL e LR) 17](#_Toc195921477)

[🔼 LL Parsing (Top-Down) 17](#_Toc195921478)

[🔽 LR Parsing (Bottom-Up) 17](#_Toc195921479)

[📚 Resumo visual 18](#_Toc195921480)

[🧪 Desafios para ti: 18](#_Toc195921481)

[🧱 4. **Autómatos com Pilha (PDA)** 19](#_Toc195921482)

[📌 Introdução: O que são PDAs? 19](#_Toc195921483)

[🧭 4.1 Ligação com Linguagens Livre de Contexto 19](#_Toc195921484)

[🔁 Diferença entre PDA e DFA 19](#_Toc195921485)

[📐 4.2 Definição formal de um PDA 19](#_Toc195921486)

[📥 4.3 Exemplo prático 20](#_Toc195921487)

[🧮 4.4 Equivalência entre PDA e CFG 20](#_Toc195921488)

[⬅️ CFG → PDA 20](#_Toc195921489)

[➡️ PDA → CFG 20](#_Toc195921490)

[🧩 4.5 Modelação de expressões com parêntesis e recursividade 21](#_Toc195921491)

[📚 Exemplo: Linguagem dos parêntesis bem formados 21](#_Toc195921492)

[🧠 Outras linguagens típicas de PDA 21](#_Toc195921493)

[📊 Resumo visual 21](#_Toc195921494)

[🧪 Desafios para ti: 22](#_Toc195921495)

[🖥️ 5. **Máquinas de Turing (MT)** 23](#_Toc195921496)

[📌 O que são? 23](#_Toc195921497)

[🧱 5.1 Definição formal e exemplos ilustrativos 23](#_Toc195921498)

[✍️ Uma MT é uma séptupla: 23](#_Toc195921499)

[📏 Estrutura da máquina 23](#_Toc195921500)

[🔄 Funcionamento (passo a passo) 23](#_Toc195921501)

[📋 Exemplo simples 24](#_Toc195921502)

[✅ 5.2 Máquina como aceitador e como transdutor 24](#_Toc195921503)

[📥 MT como **Aceitador** (Reconhecedor de Linguagem) 24](#_Toc195921504)

[🔄 MT como **Transdutor** (Máquina de computação) 24](#_Toc195921505)

[🌍 5.3 Importância como modelo universal de computação 24](#_Toc195921506)

[🧬 5.4 Variantes da Máquina de Turing 25](#_Toc195921507)

[📄 1. **MT de múltiplas fitas** 25](#_Toc195921508)

[🔀 2. **MT não-determinista** 25](#_Toc195921509)

[🧠 3. **MT Universal** 25](#_Toc195921510)

[♾️ 4. **MT com fita bi-infinita** 25](#_Toc195921511)

[📊 Resumo visual 26](#_Toc195921512)

[🧪 Desafios para ti: 26](#_Toc195921513)

[🧠 6. **Decidibilidade e Indecidibilidade** 27](#_Toc195921514)

[⚖️ 6.1 O que é um problema decidível? 27](#_Toc195921515)

[🚫 6.2 O Halting Problem e problemas indecidíveis 27](#_Toc195921516)

[🔄 6.3 Problemas indecidíveis famosos 28](#_Toc195921517)

[📬 6.4 Post Correspondence Problem (PCP) 28](#_Toc195921518)

[📐 6.5 Lógica computacional e os limites da computação 28](#_Toc195921519)

[🧠 O Teorema de Rice 29](#_Toc195921520)

[📊 Resumo visual 29](#_Toc195921521)

[🧪 Desafios para ti: 29](#_Toc195921522)

[🧠 7. **Teoremas Fundamentais da Teoria da Computação** 30](#_Toc195921523)

[📏 7.1 **Teorema de Rice** 30](#_Toc195921524)

[🧩 7.2 **Teorema de Myhill–Nerode** 30](#_Toc195921525)

[🧠 7.3 **Teorema de Cook-Levin** 31](#_Toc195921526)

[📐 7.4 **Diagonalização de Cantor** 31](#_Toc195921527)

[🧠 Resumo Visual 32](#_Toc195921528)

[🧮 8. **Funções Recursivas e Modelos Alternativos de Computação** 33](#_Toc195921529)

[🔢 8.1 Funções primitivas e μ-recursivas 33](#_Toc195921530)

[💥 8.2 A função de Ackermann 34](#_Toc195921531)

[🧠 8.3 O λ-cálculo e os números de Gödel 34](#_Toc195921532)

[🔁 8.4 Sistemas de reescrita: Post, Markov, L-Systems 34](#_Toc195921533)

[📊 Quadro comparativo 35](#_Toc195921534)

[🧠 Conclusão 36](#_Toc195921535)

[⏱️ 9. **Complexidade Computacional** 37](#_Toc195921536)

[🔹 9.1 Classes P, NP, NP-Completo, NP-Difícil 37](#_Toc195921537)

[📉 9.2 Reduções Polinomiais 38](#_Toc195921538)

[💾 9.3 Complexidade de espaço: PSPACE, L, NL 38](#_Toc195921539)

[🧠 9.4 Teoremas e Problemas Clássicos 39](#_Toc195921540)

[📊 Resumo visual das classes 40](#_Toc195921541)

[🧪 Desafios para ti 40](#_Toc195921542)

[🧬 **10. Hierarquia de Linguagens e Modelos de Computação** 41](#_Toc195921543)

[🔹 10.1 Hierarquia de Chomsky: 41](#_Toc195921544)

[🔹 10.2 Gramáticas Contextuais e Não-Restritas 41](#_Toc195921545)

[🔹 10.3 LBA – Autómatos Linearmente Limitados 42](#_Toc195921546)

[🔹 10.4 Comparação de Poderes Computacionais 42](#_Toc195921547)

[📚 Exemplo visual da Hierarquia 43](#_Toc195921548)

[🌟 Aplicações e Reflexões 43](#_Toc195921549)

[🧪 Desafios práticos 43](#_Toc195921550)

### 🎯 **Os 10 Tópicos Essenciais em Teoria da Computação**

#### 1. **Linguagens Regulares e Autómatos Finitos**

🔹 Definições formais de DFA e NFA  
🔹 Equivalência entre DFA, NFA e expressões regulares  
🔹 Conversão e minimização de autómatos  
🔹 Importância na análise léxica de compiladores

#### 2. **Expressões Regulares e Gramáticas Regulares**

🔹 Potência expressiva das REs  
🔹 Correspondência com autómatos  
🔹 Algoritmos de conversão entre REs e AFDs  
🔹 Papel crucial em motores de busca e validação de padrões

#### 3. **Gramáticas Livre de Contexto (CFG) e Análise Sintática**

🔹 Derivações, árvores sintáticas, ambiguidade  
🔹 Normalizações (Chomsky, Greibach)  
🔹 Importância nas linguagens de programação  
🔹 Técnicas de parsing (LL, LR)

#### 4. **Autómatos com Pilha (PDA)**

🔹 Ligação com linguagens livre de contexto  
🔹 Equivalência entre PDA e CFG  
🔹 Modelação de expressões com parêntesis, recursividade

#### 5. **Máquinas de Turing (MT)**

🔹 Definição formal e exemplos ilustrativos  
🔹 Máquina como aceitador e como transdutor  
🔹 Importância como modelo universal de computação  
🔹 Variantes: multi-fita, não-determinística, universal

#### 6. **Decidibilidade e Indecidibilidade**

🔹 Problemas decidíveis com MT  
🔹 Halting problem e problemas indecidíveis  
🔹 Post Correspondence Problem  
🔹 Lógica computacional e limites da computação

#### 7. **Teoremas Fundamentais**

🔹 Teorema de Rice  
🔹 Teorema de Myhill-Nerode  
🔹 Teorema de Cook-Levin (NP-completude)  
🔹 Diagonalização de Cantor

#### 8. **Funções Recursivas e Modelos Alternativos**

🔹 Funções primitivas, μ-recursivas  
🔹 Função de Ackermann  
🔹 λ-cálculo, números de Gödel  
🔹 Sistemas de reescrita: Post, Markov, L-Systems

#### 9. **Complexidade Computacional**

🔹 Classes P, NP, NP-completo, NP-difícil  
🔹 Reduções polinomiais  
🔹 Espaço e classes como PSPACE, L, NL  
🔹 Savitch, TQBF, Generalized Geography

#### 10. **Hierarquia de Linguagens e Modelos de Computação**

🔹 Hierarquia de Chomsky: Regular ⊂ CFL ⊂ CSL ⊂ Recursively Enumerable  
🔹 Gramáticas contextuais e não-restritas  
🔹 LBA (autómatos linearmente limitados)  
🔹 Comparação de poderes computacionais

## 🔍 1. **Linguagens Regulares e Autómatos Finitos**

### 🧠 Visão Geral

As **linguagens regulares** e os **autómatos finitos** representam o **nível mais simples e eficiente de reconhecimento automático de padrões**. São a base para tarefas computacionais como a **análise léxica de compiladores**, **validação de strings**, e até **motores de pesquisa com expressões regulares**.

## 🧩 1.1 Definições formais de DFA e NFA

### 📘 DFA – Autómato Finito Determinista

Um DFA é uma quíntupla:

* : conjunto finito de estados
* : alfabeto de entrada
* : função de transição
* : estado inicial
* : conjunto de estados finais

✅ *Para cada par (estado, símbolo), existe* ***uma única*** *transição possível.*

### 🎲 NFA – Autómato Finito Não Determinista

Também definido por uma quíntupla, mas com:

* Aqui, uma entrada pode levar a **vários estados**, ou **nenhum**.
* Permite computações "paralelas", conceptualizadas como **exploração de múltiplos caminhos** ao mesmo tempo.

🧪 Também existe o **NFA-ε**, que permite **transições espontâneas** (sem consumir símbolo).

## 🔁 1.2 Equivalência entre DFA, NFA e Expressões Regulares

Um dos resultados mais belos e poderosos da teoria:

**Toda linguagem reconhecida por um DFA pode ser reconhecida por um NFA, e vice-versa.**

Além disso, **toda linguagem regular pode ser descrita por uma expressão regular**.

🔄 Transformações importantes:

* **NFA → DFA**: via o **método de subconjuntos** (construção do autómato de estados de conjuntos).
* **AF → RE**: usando o **algoritmo de Arden**, ou eliminação de estados.
* **RE → NFA-ε**: construção direta a partir dos operadores |, ., \*.

📚 Isto demonstra a **equivalência formal entre RE, DFA, NFA e gramáticas regulares**.

## 🧼 1.3 Minimização de Autómatos

### 🧮 O problema:

Dado um DFA, pode haver **estados redundantes** que reconhecem exatamente a mesma linguagem.

### ✂️ A solução:

* Identificar **estados equivalentes** (Myhill-Nerode)
* Fundir esses estados
* Produzir o **DFA mínimo**, com o menor número de estados possível

✅ A minimização é **determinística e única até isomorfismo**.

## 🧠 1.4 Importância prática: Análise Léxica de Compiladores

Durante a **análise léxica** (ou *scanning*) de código fonte:

1. **Tokens** (palavras-chave, identificadores, números, símbolos) são extraídos com base em **padrões regulares**
2. Cada token é reconhecido por um **DFA gerado a partir de uma RE**
3. O **analisador léxico** (frequentemente gerado com ferramentas como Lex ou Flex) usa esses autómatos para **varrer o código e produzir uma sequência de tokens**

💻 Exemplo prático:

if (x == 42) { return true; }

Tokens:  
→ if, (, id, ==, num, ), {, return, true, ;, }

✅ Cada token é **reconhecido por um DFA**, construído a partir de **expressões regulares definidas pelo programador**.

## 🎯 Conclusão e Ligações Pedagógicas

Este tema é **essencial** porque combina:

* **Conceitos formais rigorosos**
* **Aplicações práticas reais**
* **Algoritmos eficientes**
* **Intuições visuais** (diagramas de estado)

### 💡 Atividades sugeridas:

1. **Desenha um DFA** que reconheça strings binárias que terminem em 01.
2. **Transforma uma RE** como (a|b)\*abb num NFA-ε e depois num DFA.
3. **Minimiza um DFA** com 7 estados que reconheça múltiplos de 3 em binário.
4. **Explora visualmente** como Flex transforma REs em código C com autómatos embutidos.

## 🔤 2. **Expressões Regulares e Gramáticas Regulares**

### 💬 O que são Expressões Regulares?

As **expressões regulares (REs)** são uma notação formal usada para **descrever linguagens regulares**, ou seja, **conjuntos de strings reconhecíveis por autómatos finitos**.

## 📏 2.1 Potência expressiva das REs

As REs são **muito expressivas dentro do universo das linguagens regulares**, mas **não vão além disso**. São ideais para:

* Padrões com **sequência fixa ou variável**
* **Repetição finita ou infinita** de símbolos
* **Escolha entre alternativas**

💡 A linguagem definida por uma RE é sempre **regular**, o que significa que pode ser reconhecida por **um DFA** (Autómato Finito Determinista).

### 🎨 Operadores fundamentais:

| Operador | Significado | Exemplo |
| --- | --- | --- |
| a | símbolo literal | aceita apenas "a" |
| `a | b` | escolha (ou) |
| ab | concatenação | "a" seguido de "b" |
| a\* | zero ou mais repetições | "", "a", "aaa" |
| a+ | uma ou mais repetições | "a", "aa", ... |
| (abc) | agrupamento | "abc" |

## 🔄 2.2 Correspondência com Autómatos

Um dos pilares da Teoria da Computação:

Para **cada expressão regular**, existe um **AFN (ou DFA)** que reconhece a mesma linguagem, e vice-versa.

🔁 Isto implica que:

* As **linguagens descritas por REs são as linguagens regulares**
* Podemos **converter REs em autómatos** e **vice-versa**

### 🎓 Implicações práticas:

| Representação | Equivalente |
| --- | --- |
| Expressão regular | AFN (via construção direta) |
| AFN | DFA (via algoritmo de subconjuntos) |
| DFA | Expressão regular (via eliminação de estados ou método de Arden) |

🧠 Esta equivalência é um **triângulo perfeito**: RE ↔ AFN ↔ DFA

## 🔧 2.3 Algoritmos de Conversão entre REs e AFDs

### 🛠️ RE → NFA-ε (construção direta)

Para cada operador:

* a: cria-se um autómato com uma transição de a
* a|b: cria-se um novo estado inicial e final, com transições via ε
* ab: encadeamento de autómatos
* a\*: adiciona ciclos com transições ε

### 📘 NFA → DFA (algoritmo de subconjuntos)

* Cada estado do DFA representa um **conjunto de estados** do NFA
* Estado inicial do DFA: ε-fecho do estado inicial do NFA
* Transições baseadas em todos os símbolos e possíveis destinos

### ♻️ DFA → RE (eliminação de estados)

* Remove-se estados gradualmente
* Substituem-se transições por **expressões equivalentes**
* Resultado: uma RE que representa o comportamento do DFA

## 🌍 2.4 Papel crucial em motores de busca e validação de padrões

As REs são **amplamente utilizadas na prática**, em múltiplas linguagens e ferramentas:

| Área | Exemplo de uso |
| --- | --- |
| **Pesquisa textual** | grep, sed, awk, regex em Python, Java, etc. |
| **Compiladores** | Definição de tokens léxicos com REs (usadas por Flex) |
| **Validação de dados** | Verificar emails, NIFs, datas, passwords |
| **Editores de texto** | Sublime, VS Code, Notepad++, todos suportam regex |
| **Análise de logs** | Extrair padrões em Apache, NGINX, sistemas operativos |

### 💡 Exemplos úteis do mundo real:

1. **Email válido**:

^[a-z0-9.\_%+-]+@[a-z0-9.-]+\.[a-z]{2,}$

1. **Número binário par**:

(0|1)\*0

1. **Identificador de variável**:

[a-zA-Z\_][a-zA-Z0-9\_]\*

## 📌 Resumo visual

| Conceito | Ligação |
| --- | --- |
| RE | Forma de definir linguagens regulares |
| AFN | Reconhece a linguagem da RE |
| DFA | Versão determinística do AFN |
| Expressividade | Apenas linguagens regulares |
| Conversões | RE → AFN → DFA → RE (ciclo completo) |
| Aplicações práticas | Motores de busca, compiladores, validação |

## 🧪 Desafios para ti:

1. **Cria uma RE** que aceite qualquer string binária que comece e termine com 1.
2. **Desenha o NFA-ε** correspondente à RE (a|b)\*abb.
3. **Verifica com grep** ou re do Python se uma password é segura (min 8 chars, letras + números).
4. **Explora visualmente** a conversão RE → AFD com diagramas de estado.

## 🧾 3. **Gramáticas Livre de Contexto (CFG) e Análise Sintática**

### 📘 3.1 O que é uma CFG?

Uma **CFG (Context-Free Grammar)** é um sistema formal para gerar linguagens, especialmente útil para **estruturas aninhadas** como expressões aritméticas, blocos de código ou estruturas XML.

É uma **quíntupla**:

* : variáveis (símbolos não-terminais)
* : alfabeto (símbolos terminais)
* : conjunto de regras de produção do tipo , onde e
* : símbolo inicial

💡 Uma CFG **gera strings** através de **substituições sucessivas**, começando pelo símbolo inicial.

## 🌳 3.2 Derivações, Árvores Sintáticas e Ambiguidade

### 🔁 Derivações

Uma **derivação** é uma sequência de aplicações de regras da CFG. Pode ser:

* **À esquerda**: substitui-se sempre o **não-terminal mais à esquerda**
* **À direita**: substitui-se sempre o **mais à direita**

### 🌲 Árvores Sintáticas (Parse Trees)

Representam **visualmente** a forma como a string é derivada:

* Raiz: símbolo inicial
* Ramos: regras aplicadas
* Folhas: símbolos terminais que compõem a string

#### Exemplo:

Para a gramática:

E → E + E | num

A árvore para num + num + num **pode ser construída de várias formas** — e isso leva-nos à…

### ⚠️ Ambiguidade

Uma gramática é **ambígua** se existe **mais do que uma árvore de derivação para a mesma string**.

⚠️ Isto é **indesejável** em linguagens de programação, pois gera **interpretações contraditórias**.

💡 Algumas linguagens **não têm gramáticas livres de contexto não-ambíguas**, mas na prática tenta-se sempre evitar a ambiguidade através de **refinamento da gramática**.

## 📏 3.3 Normalizações (Chomsky e Greibach)

### 🔹 Forma Normal de Chomsky (CNF)

Cada produção tem uma das formas:

💡 Todas as CFGs **sem ε-produções ou recursividade direta** podem ser convertidas em CNF.

**Usos:**

* Algoritmo de Cocke–Younger–Kasami (CYK)
* Análise sintática eficiente e bottom-up

### 🔹 Forma Normal de Greibach (GNF)

Cada produção tem a forma:

💡 Toda CFG pode ser convertida em GNF (com algum esforço).  
Essencial para parsing **top-down**, pois garante que a **entrada é consumida imediatamente**.

## 🧠 3.4 Importância nas Linguagens de Programação

A maioria das linguagens de programação tem **gramáticas livre de contexto** (ou ligeiramente context-sensitive na semântica).

➡️ As CFGs são **fundamentais para o parsing**, que verifica se o programa **tem estrutura válida** antes de o compilar.

#### Exemplo:

if (x < 5) { return x + 1; }

Pode ser analisado com uma CFG simplificada do tipo:

stmt → if ( cond ) stmt  
cond → expr < expr  
expr → expr + expr | id | num

🎯 Cada linha de código é validada como **derivável pela gramática da linguagem**.

## 🔍 3.5 Técnicas de Parsing (LL e LR)

### 🔼 LL Parsing (Top-Down)

* **L**eft-to-right, **L**eftmost derivation
* Usa **tabela preditiva**
* Exige gramáticas **fatoradas** e **sem recursividade à esquerda**

**LL(1)**:

* Lê 1 símbolo à frente
* Fácil de implementar
* Base de parsers recursivos (ex: ANTLR)

### 🔽 LR Parsing (Bottom-Up)

* **L**eft-to-right, **R**ightmost derivation (reconstruída)
* Muito mais **poderoso**
* Usa **autómatos com pilha + tabela de parsing**
* Lida com maior classe de gramáticas

Tipos:

* **LR(0)**: básico, sem lookahead
* **SLR(1)**: usa conjuntos Follow
* **LALR(1)**: combinação prática, usada em Bison/Yacc
* **Canonical LR(1)**: mais preciso, mas complexo

✅ A maioria dos **compiladores reais usa parsers LR**, especialmente **LALR(1)**.

## 📚 Resumo visual

| Conceito | Explicação |
| --- | --- |
| CFG | Regras para gerar linguagens LFC |
| Derivação | Sequência de aplicações de regras |
| Parse Tree | Estrutura hierárquica da derivação |
| Ambiguidade | Múltiplas árvores para a mesma string |
| CNF | Forma normal para algoritmos e análise |
| GNF | Forma útil para parsing top-down |
| LL | Parsing preditivo (top-down) |
| LR | Parsing baseado em pilha (bottom-up) |

## 🧪 Desafios para ti:

1. Escreve uma CFG para expressões como a + b \* c.
2. Constrói duas árvores sintáticas diferentes para a string num + num + num.
3. Transforma uma CFG simples em CNF.
4. Implementa um **parser LL(1)** em Python para expressões aritméticas.
5. Usa o Bison para gerar um **parser LALR(1)** para comandos if/else.

## 🧱 4. **Autómatos com Pilha (PDA)**

### 📌 Introdução: O que são PDAs?

Um **PDA (Pushdown Automaton)** é um autómato finito que tem acesso a uma **pilha** — uma estrutura LIFO (Last In, First Out), onde se pode:

* **Empilhar (push)** um símbolo
* **Desempilhar (pop)** o topo
* **Verificar** o símbolo no topo

💡 Esta pilha confere ao PDA uma **memória limitada mas poderosa**, suficiente para lidar com **estruturas recursivas e aninhadas**.

## 🧭 4.1 Ligação com Linguagens Livre de Contexto

A relação é direta e profunda:

**Uma linguagem é livre de contexto ⇔ Existe um PDA que a reconhece**

Isto significa que:

* Toda **CFG tem um PDA equivalente**
* Toda linguagem que pode ser descrita por uma CFG **pode ser reconhecida por um PDA**

💬 **Padrões recursivos**, como expressões com parêntesis ou estruturas tipo “matrioska”, são exemplos típicos de linguagens LFC, que requerem memória de pilha para serem reconhecidos corretamente.

### 🔁 Diferença entre PDA e DFA

| DFA | PDA |
| --- | --- |
| Sem memória | Usa pilha (memória limitada) |
| Reconhece linguagens regulares | Reconhece linguagens livre de contexto |
| Cada símbolo decide estado seguinte | Decisão depende também do **símbolo no topo da pilha** |

## 📐 4.2 Definição formal de um PDA

Um PDA é uma **séptupla**:

* : conjunto de estados
* : alfabeto de entrada
* : alfabeto da pilha
* : função de transição
* : estado inicial
* : símbolo inicial da pilha
* : conjunto de estados finais

A máquina pode **ler ou não** um símbolo da entrada, e também pode:

* **Consultar** o topo da pilha
* **Substituí-lo** por zero ou mais símbolos

### 📥 4.3 Exemplo prático

Reconhecer a linguagem :

**Ideia:**

* Empilhar a enquanto os lemos
* Para cada b, desempilhar um a
* Aceitar se a pilha estiver vazia no fim

## 🧮 4.4 Equivalência entre PDA e CFG

### ⬅️ CFG → PDA

Para cada **produção da CFG**, construímos **transições do PDA** que:

* Lêem o topo da pilha como uma variável não-terminal
* Substituem-no pela sequência da produção (regras da gramática)
* Continuam até que a pilha contenha apenas terminais

🧠 É como o PDA **simular a derivação de uma CFG**.

### ➡️ PDA → CFG

Para cada transição do PDA, criamos **regras da CFG** que representam os efeitos da pilha e os símbolos lidos.

💡 Este processo é mais técnico, mas **garante que toda linguagem reconhecida por um PDA tem uma CFG correspondente**.

## 🧩 4.5 Modelação de expressões com parêntesis e recursividade

As expressões com **aninhamentos recursivos** não podem ser reconhecidas por DFA. Porquê? Porque o DFA **não consegue contar ou lembrar quantos símbolos abriu**.

### 📚 Exemplo: Linguagem dos parêntesis bem formados

**Solução com PDA:**

* Quando lê (: empilha
* Quando lê ): desempilha
* Aceita se a pilha estiver vazia no final e todos os símbolos tiverem sido lidos

📌 Este tipo de comportamento é **impossível para DFAs**, mas **trivial para PDAs**.

### 🧠 Outras linguagens típicas de PDA

| Linguagem | Justificação |
| --- | --- |
|  | Requer contagem (via pilha) |
| Expressões aritméticas com parêntesis | Aninhamento |
| Blocos de código (if/else, begin/end) | Estrutura recursiva |
| XML/HTML bem formados | Abertura/fecho de tags |

## 📊 Resumo visual

| Conceito | Significado |
| --- | --- |
| PDA | Autómato com memória tipo pilha |
| CFG | Gramática para linguagens livre de contexto |
| Equivalência | Toda CFG tem um PDA correspondente e vice-versa |
| Parêntesis | Modelados com empilhar/desempilhar |
| Recursividade | Naturalmente representada via pilha |

## 🧪 Desafios para ti:

1. Cria um PDA para reconhecer
2. Transforma a CFG num PDA equivalente
3. Escreve uma CFG para expressões aritméticas com +, \*, e parêntesis
4. Usa um simulador (ex: [PDA Simulator](https://automata.cs.luc.edu)) para ver o comportamento em tempo real

## 🖥️ 5. **Máquinas de Turing (MT)**

### 📌 O que são?

A **Máquina de Turing** é um modelo matemático abstrato proposto para **formalizar a noção de algoritmo**.

Apesar da sua simplicidade, ela é **tão poderosa quanto qualquer computador moderno**, no sentido de que tudo o que se pode programar *em princípio* pode ser modelado por uma MT.

## 🧱 5.1 Definição formal e exemplos ilustrativos

### ✍️ Uma MT é uma séptupla:

* : conjunto finito de estados
* : alfabeto de entrada (não inclui branco ␣)
* : alfabeto da fita (inclui ␣)
* : função de transição
* : estado inicial
* : estado de aceitação
* : estado de rejeição

### 📏 Estrutura da máquina

* Uma **fita infinita** (com células que armazenam símbolos)
* Uma **cabeça de leitura/escrita** que se move para **esquerda ou direita**
* Um **estado atual**, que guia a computação
* Um **alfabeto de trabalho**, incluindo um símbolo branco ␣

### 🔄 Funcionamento (passo a passo)

1. Lê o símbolo sob a cabeça
2. A função decide:
   * Novo símbolo a escrever
   * Direção a mover (L ou R)
   * Novo estado
3. O processo repete-se até entrar em ou

### 📋 Exemplo simples

Linguagem

**Estratégia**:

* Substitui o primeiro a por X
* Varre à direita e substitui o primeiro b por Y
* Volta à esquerda para o próximo a
* Repete até que só haja X e Y
* Aceita se todos os a e b forem emparelhados

Este comportamento **exige memória arbitrária**, algo impossível para DFAs ou PDAs.

## ✅ 5.2 Máquina como aceitador e como transdutor

### 📥 MT como **Aceitador** (Reconhecedor de Linguagem)

* A fita contém uma string de entrada
* A máquina aceita se, ao processar a string, entra em
* Exemplo: reconhecer strings palíndromas, divisíveis por 3, expressões bem formadas

### 🔄 MT como **Transdutor** (Máquina de computação)

* A fita é usada para realizar **cálculos**
* A saída é o conteúdo da fita no final da execução
* Exemplo: somar dois números binários representados em fita

💬 Neste caso, a MT **não está a reconhecer uma linguagem**, mas a **transformar uma entrada numa saída**.

## 🌍 5.3 Importância como modelo universal de computação

A MT é o **modelo canónico de algoritmo**. Tudo o que é computável, é computável por uma MT.

Esta ideia está na base do **Tese de Church-Turing**:

*“Qualquer função computável por um algoritmo pode ser computada por uma Máquina de Turing.”*

💡 Implicações:

* As linguagens que **MTs conseguem reconhecer** são chamadas **recursivamente enumeráveis**
* As que são **decidíveis** (MT termina sempre) são chamadas **recursivas**
* Estabelece o **limite teórico do que é computável**

## 🧬 5.4 Variantes da Máquina de Turing

Apesar da simplicidade do modelo base, várias **variações poderosas** foram propostas — todas **equivalentes em poder computacional**.

### 📄 1. **MT de múltiplas fitas**

* Várias fitas paralelas, cada uma com a sua cabeça
* Mais fácil de programar
* Pode ser simulada por uma MT de fita única com codificação adequada

### 🔀 2. **MT não-determinista**

* A função pode devolver **vários resultados**
* A computação **explora vários caminhos simultaneamente**
* Aceita uma string se **algum caminho levar a**

💡 Surpreendentemente:

**MT deterministas e não-deterministas são equivalentes em poder computacional** (mas não necessariamente em tempo).

### 🧠 3. **MT Universal**

* Uma MT que pode **simular qualquer outra** MT, dado o seu código e a sua entrada
* Idêntico ao conceito de **computador moderno** com programas armazenados

A MT universal é **a base teórica da programação**.

### ♾️ 4. **MT com fita bi-infinita**

* A fita estende-se infinitamente em **ambas as direções**
* Torna certas construções mais simples
* Equivalente em poder à MT tradicional

## 📊 Resumo visual

| Variante | Diferença | Poder computacional |
| --- | --- | --- |
| MT básica | Uma fita, determinista | Total |
| MT multi-fita | Várias fitas | Igual |
| MT não-determinista | Vários ramos possíveis | Igual |
| MT universal | Simula outras MTs | Igual |
| MT bi-infinita | Fita para os dois lados | Igual |

## 🧪 Desafios para ti:

1. Escreve um algoritmo (em pseudo-MT) para inverter uma string binária.
2. Desenha uma MT que aceita strings do tipo (duplicação).
3. Simula uma MT universal com código e entrada codificados.
4. Compara o tempo de execução de uma tarefa com 1 fita vs. 2 fitas.

## 🧠 6. **Decidibilidade e Indecidibilidade**

### ⚖️ 6.1 O que é um problema decidível?

Um problema é **decidível** (ou **resolúvel**) se existe uma **Máquina de Turing que termina sempre**, para qualquer entrada, dizendo “sim” ou “não”.

🔁 A MT **não entra em loop** — devolve sempre uma resposta correta.

#### 🟢 Exemplos clássicos de problemas decidíveis:

| Problema | Estratégia de resolução |
| --- | --- |
| Se um DFA aceita uma string | Simular o DFA (termina sempre) |
| Se um DFA reconhece linguagem vazia | Verificar acessibilidade de estados finais |
| Se uma CFG gera uma string | Usar algoritmo CYK |
| Se duas REs são equivalentes | Construir DFA mínimo e comparar |

💡 Estes problemas são resolvidos por algoritmos bem definidos, com garantia de **terminação e resposta correta**.

### 🚫 6.2 O Halting Problem e problemas indecidíveis

#### ❗ O Problema da Paragem (Halting Problem)

Dado um programa e uma entrada , será que pára ao correr ?

Formalmente:

⚠️ Este problema é **indecidível** — **nenhuma Máquina de Turing pode resolvê-lo em geral**.

#### 🧠 Prova por diagonalização (esboço)

1. Suponhamos que existe uma MT que decide
2. Usamos essa MT para construir uma nova MT que se comporta **contraditoriamente consigo própria**
3. O paradoxo leva à **contradição**
4. Logo, **nenhuma MT pode decidir**

💣 Esta ideia foi inspirada nos **teoremas da incompletude de Gödel** e é uma das mais profundas da matemática moderna.

### 🔄 6.3 Problemas indecidíveis famosos

| Problema | Descrição |
| --- | --- |
|  | Saber se uma MT pára numa entrada |
| Equivalência de MTs | Saber se duas MTs aceitam a mesma linguagem |
| Vacuidade de uma MT | Saber se a linguagem reconhecida é vazia |
| Se uma linguagem é regular ou CFL | Em geral, não decidível para MTs |

💡 Importante: **MTs podem simular qualquer algoritmo**, mas **não podem responder a tudo!**

### 📬 6.4 Post Correspondence Problem (PCP)

Um dos exemplos mais simples (e famosos) de problema indecidível, **sem recorrer a MTs diretamente**.

#### Definição:

Dado um conjunto de **pares de strings** , será que existe uma sequência de índices tal que:

🧠 Ou seja: conseguimos fazer as duas colunas “casarem” com concatenações iguais?

🔴 Não existe algoritmo geral que resolva todos os casos — o problema é indecidível.

#### Exemplo concreto:

Experimentar sequências pode parecer promissor… mas **não há maneira geral de saber se devemos continuar ou parar** — aí está o dilema da indecidibilidade!

### 📐 6.5 Lógica computacional e os limites da computação

A indecidibilidade está fortemente ligada à **teoria da lógica e das provas matemáticas**.

#### 🧮 Lógica de 1ª ordem

Determinar se uma fórmula é válida (**teorema da lógica de 1ª ordem**) é **indecidível**.

💡 Isto significa que **não existe programa que decida a verdade de todas as proposições formais**.

🔁 Este resultado está ligado ao **Teorema de Church-Turing**, que estabelece que **nem tudo é computável**.

### 🧠 O Teorema de Rice

“**Qualquer propriedade não-trivial da linguagem de uma MT é indecidível.**”

Exemplos:

* “A linguagem da MT é finita?” → indecidível
* “A linguagem é regular?” → indecidível
* “A linguagem aceita pelo programa só contém palíndromos?” → indecidível

🧨 Conclusão: **é impossível construir analisadores universais de software** que digam com certeza o que os programas fazem.

## 📊 Resumo visual

| Conceito | Tipo | Decidível? |
| --- | --- | --- |
|  | Aceitação por DFA | ✅ |
|  | Aceitação por MT | ❌ |
| Halting Problem | Terminação de MTs | ❌ |
| PCP | Igualdade por concatenação | ❌ |
| Lógica 1ª ordem | Validade de fórmulas | ❌ |
| Teorema de Rice | Propriedades semânticas | ❌ |

## 🧪 Desafios para ti:

1. Explica com as tuas palavras por que razão o problema da paragem é indecidível.
2. Dado um conjunto de pares, tenta resolver o PCP — consegues decidir ou ficas preso?
3. Imagina um "detector de bugs perfeito": como o Teorema de Rice mostra que é impossível?
4. Classifica os seguintes problemas como decidíveis ou indecidíveis:  
   a) Se uma RE gera uma linguagem vazia  
   b) Se uma MT aceita palíndromos  
   c) Se uma CFG gera aab

## 🧠 7. **Teoremas Fundamentais da Teoria da Computação**

Estes quatro teoremas são **pilares da teoria da computação**, pois estabelecem **limites, equivalências e critérios fundamentais** sobre o que pode ser computado, distinguido ou classificado.

### 📏 7.1 **Teorema de Rice**

🔹 “Qualquer propriedade não-trivial das linguagens aceites por máquinas de Turing é indecidível.”

#### 🔍 O que quer isto dizer?

Se tens uma propriedade sobre **linguagens reconhecidas por Máquinas de Turing** (como “é finita?”, “tem apenas palavras palíndromas?”, etc.) e essa propriedade **não é verdadeira para todas as linguagens nem falsa para todas**, então **não há algoritmo que a determine em todos os casos**.

#### 🎯 Implicações práticas:

* Não conseguimos construir algoritmos para **verificar propriedades semânticas de programas**.
* Não podemos, por exemplo, verificar automaticamente se um programa **vai sempre devolver um número primo**.

#### 💡 Intuição da prova:

* Usa-se uma **redução do problema da paragem** (Halting Problem), que é indecidível, para mostrar que a propriedade em questão também o é.

### 🧩 7.2 **Teorema de Myhill–Nerode**

🔹 “Uma linguagem é regular se e só se o número de classes de equivalência (em relação a uma certa relação de indistinguibilidade) é finito.”

#### 💼 Utilização:

* Ferramenta poderosa para:
  + **Provar que uma linguagem é regular**
  + **Provar que uma linguagem não é regular**
  + **Minimizar autómatos finitos deterministas (DFA)**

#### 🧠 Conceito-chave:

Dizemos que duas cadeias e são **indistinguíveis** se, para todo o sufixo ,  
$\left. xz \in L\text{\:\,}\Longleftrightarrow\text{\:\,}yz \in L \right.$

Se o número de classes de cadeias indistinguíveis for finito → a linguagem é regular.

#### 📉 Exemplo:

Para a linguagem , existe um número infinito de cadeias indistinguíveis (diferentes “n”), logo, **não é regular**.

### 🧠 7.3 **Teorema de Cook-Levin**

🔹 “O problema da satisfatibilidade booleana (SAT) é NP-completo.”

Este foi o **primeiro problema provado como NP-completo**, abrindo a porta para a **teoria da complexidade** tal como a conhecemos.

#### ✨ Consequência:

* Se **SAT pode ser resolvido em tempo polinomial**, então **todos os problemas em NP também podem!** Ou seja:

#### 💡 Intuição da prova:

Mostra-se que **qualquer problema em NP** pode ser **reduzido a uma instância de SAT**, onde se pergunta se uma fórmula booleana tem uma atribuição que a torna verdadeira.

#### 🧠 Aplicações:

* Algoritmos de **otimização, verificação de circuitos, jogos, logística**...
* É a base da ideia de **"redução polinomial"** entre problemas difíceis.

### 📐 7.4 **Diagonalização de Cantor**

🔹 Técnica desenvolvida originalmente para provar que os **números reais não são enumeráveis**, mas adaptada para mostrar que **existem linguagens não reconhecíveis por nenhuma Máquina de Turing**.

#### 🧠 Como funciona?

* Assume-se uma lista (enumerável) de máquinas de Turing.
* Constrói-se uma linguagem que, por definição, **difere de cada uma dessas máquinas em pelo menos uma entrada**.

➡️ **Conclusão:** Essa linguagem **não está na lista** → **não é reconhecível** → mostra que o conjunto de linguagens **é maior que o das MTs**.

#### 🧨 Implicação teórica:

* Existem problemas **nem sequer semi-decidíveis**!
* Mostra os **limites da computação mecânica**.

### 🧠 Resumo Visual

| Teorema | Tema Central | Implicação Principal |
| --- | --- | --- |
| Rice | Propriedades semânticas de linguagens | Indecidibilidade universal |
| Myhill-Nerode | Regularidade | Critério formal + minimização de autómatos |
| Cook-Levin | Complexidade | SAT é o “rei” dos NP-completos |
| Diagonalização de Cantor | Limites da MT | Existem linguagens nem reconhecíveis |

## 🧮 8. **Funções Recursivas e Modelos Alternativos de Computação**

Nesta parte, exploramos **formas alternativas de representar computação**, que **não dependem de máquinas de Turing**, mas sim de construções matemáticas puras, como **funções, números e regras de substituição**. Estes modelos são fundamentais para compreender os **fundamentos teóricos da computação** e os limites do que podemos calcular.

### 🔢 8.1 Funções primitivas e μ-recursivas

#### ➕ Funções primitivas recursivas

São **funções totais** (definidas para todos os inteiros naturais), construídas a partir de:

* **Funções básicas iniciais**:
  + Função zero:
  + Função sucessor:
  + Funções projeção:
* **Operações permitidas**:
  + Composição de funções
  + Recursão primitiva (definição em termos de valores anteriores)

✅ Estas funções **são sempre computáveis** e **terminam sempre**. Contudo, **não conseguem expressar tudo**, por exemplo, a **função de Ackermann**.

#### 🌀 Funções μ-recursivas

Estas expandem as primitivas com a operação de **minimização (μ)**, permitindo definir funções **parciais** (que podem não terminar).

$$\mu y\text{\:\,}\lbrack f(x,y) = 0\rbrack$$

É o primeiro valor tal que , **se existir**.

⚠️ A operação μ introduz **não-terminação** → estas funções são **equivalentes às computáveis por Máquinas de Turing**.

### 💥 8.2 A função de Ackermann

É um clássico exemplo de função **computável mas não primitiva recursiva**.

#### Definição:

📈 Cresce **mais depressa do que qualquer função primitiva recursiva**.  
💡 Mostra que **nem todas as funções computáveis podem ser obtidas por recursão primitiva**.

### 🧠 8.3 O λ-cálculo e os números de Gödel

#### 🧮 λ-Cálculo

Criado por Alonzo Church, é um sistema formal minimalista baseado em **funções e aplicação**.

* Exemplo: representa uma função anónima que soma 1
* Tem **poder computacional equivalente a Turing**

🔁 Todas as computações podem ser representadas por **aplicações sucessivas de funções**.  
🔁 Importante na base teórica das **linguagens funcionais modernas** (Haskell, Lisp, etc.)

#### 🔢 Números de Gödel

Cada expressão formal pode ser codificada por um **número inteiro único**.  
Exemplo:

* Símbolos são mapeados para primos:
* Uma sequência torna-se

💡 Usado na prova do **Teorema da Incompletude** de Gödel  
💡 Fundamenta a **auto-referência na computação** (por ex., o Teorema da Recursão)

### 🔁 8.4 Sistemas de reescrita: Post, Markov, L-Systems

#### 📬 Sistemas de Post

* Baseiam-se em **pares de regras de substituição**
* Funcionam como:  
  “Se vês esta sequência, substitui por esta outra”

🔁 São **equivalentes a Máquinas de Turing** em poder computacional  
💡 Muito usados para ilustrar **transformações simbólicas**

#### 🧮 Algoritmos de Markov

* Similares aos sistemas de Post, mas com **regras ordenadas**
* Apenas uma regra se aplica por iteração: a **primeira aplicável**

✅ Também **computacionalmente universais**  
✅ Usados em **geração de texto e IA primitiva**

#### 🌿 L-Systems (Linguagens de Lindenmayer)

* Usados para **modelar crescimento de plantas**
* Cada símbolo é substituído simultaneamente segundo regras (reescrita paralela)

🎨 Muito usados em computação gráfica, biologia e simulação do crescimento vegetal  
🔁 Tecnicamente, são **sistemas formais não universais**, mas ilustrativos e visuais

### 📊 Quadro comparativo

| Modelo | Poder computacional | Terminação garantida? | Aplicações |
| --- | --- | --- | --- |
| Funções primitivas | Menor que Turing | ✅ | Matemática básica |
| μ-recursivas | Igual à MT | ❌ | Teoria da computação |
| λ-cálculo | Igual à MT | ❌ | Linguagens funcionais |
| Post/Markov | Igual à MT | ❌ | Teoria formal, IA |
| L-Systems | Menor que MT | ✅ | Biologia, gráficos |

## 🧠 Conclusão

A diversidade de modelos mostra que:

* A **ideia de computação** não depende de uma máquina em si, mas de **operações formais universais**
* **Todas as funções computáveis** podem ser expressas por uma **MT, μ-recursivas ou λ-cálculo**
* Modelos como **L-Systems e Post** ajudam-nos a **visualizar e aplicar** estes conceitos em áreas inesperadas

## ⏱️ 9. **Complexidade Computacional**

### 🔹 9.1 Classes P, NP, NP-Completo, NP-Difícil

#### ✅ **Classe P** (Polynomial Time)

* Conjunto de problemas **decidíveis em tempo polinomial** por uma **MT determinista**
* Considerados “**eficientemente resolvíveis**”

💡 Exemplos:

* Ordenação (Merge Sort: )
* Verificação de grafos conexos
* Multiplicação de matrizes

#### ❓ **Classe NP** (Nondeterministic Polynomial Time)

* Problemas cujas **soluções podem ser verificadas** em tempo polinomial
* Ou: problemas decidíveis por uma **MT não-determinista em tempo polinomial**

💡 Exemplo clássico:  
**Sudoku** → Fácil de verificar, difícil de resolver

#### 🔁 **NP-completos**

* Problemas em NP para os quais **qualquer outro problema em NP pode ser reduzido em tempo polinomial**
* São os **mais difíceis de NP**

💣 Se resolveres **um** deles em tempo polinomial, resolves **todos os NP** em tempo polinomial!

🔁 Exemplo:  
**SAT** (problema da satisfatibilidade booleana) — o **primeiro problema provado como NP-completo (Teorema de Cook-Levin)**

#### 💥 **NP-difícil**

* Problemas **tão difíceis como os NP-completos**, mas **não precisam de estar em NP**
* Podem ser **não-decidíveis**

💡 Exemplo: **TSP (Traveling Salesman Problem)** com minimização de percurso

### 📉 9.2 Reduções Polinomiais

Se consigo transformar **qualquer instância de um problema A num problema B** em tempo polinomial, e resolver B, então **consigo resolver A**.

#### ⚙️ Usos principais:

* **Provar que um problema é NP-completo** (mostrar que um NP-completo se reduz a ele)
* **Estudar hierarquias de dificuldade**

#### Notação:

→ A é redutível a B em tempo polinomial

💡 Consequência: Se B está em P, então A também está.

### 💾 9.3 Complexidade de espaço: PSPACE, L, NL

#### 💿 **PSPACE** (Polynomial Space)

* Problemas que podem ser **resolvidos com memória polinomial**
* **Não importa o tempo**, apenas o espaço consumido

💡 Inclui **P** e **NP**, mas pode conter problemas **ainda mais difíceis**

#### 💾 **L** (Logspace)

* Usa apenas **memória proporcional a**
* Muito restrito — ex: verificar se uma lista está ordenada com pouco espaço

#### 💽 **NL** (Nondeterministic Logspace)

* Como L, mas com não-determinismo
* Exemplo clássico: **Reachability** (existe caminho entre dois vértices?)

#### 🔁 Relações conhecidas:

ℹ️ Nem todas estas inclusões são sabidas como estritas — **por exemplo, ainda não sabemos se P = PSPACE**.

### 🧠 9.4 Teoremas e Problemas Clássicos

#### 🧠 **Savitch’s Theorem**

“Qualquer problema resolvido com não-determinismo e espaço , pode ser resolvido com determinismo e espaço .”

Implicação direta:

💡 Mostra que o **não-determinismo em espaço é menos poderoso que em tempo**

#### 🧠 **TQBF (Totally Quantified Boolean Formula)**

* Fórmulas booleanas com quantificadores alternados:
* Problema **PSPACE-completo** → o mais difícil da classe PSPACE

Usado para provar que outros problemas estão em PSPACE.

#### 🧠 **Generalized Geography**

* Jogo sobre grafos onde jogadores alternam movendo-se para vértices não visitados
* Pergunta: **o primeiro jogador tem estratégia vencedora?**

🔁 Este problema é também **PSPACE-completo**

💡 Mostra que até **jogos simples** podem ser **incrivelmente difíceis do ponto de vista computacional**

### 📊 Resumo visual das classes

| Classe | Recurso | Exemplo | Complexidade |
| --- | --- | --- | --- |
| **P** | Tempo determinista polinomial | Soma de inteiros | Baixa |
| **NP** | Tempo não-determinista polinomial | Sudoku | Alta |
| **NP-completo** | Mais difícil de NP | SAT, 3-COL | Muito alta |
| **NP-difícil** | Fora de NP, mas comparável | TSP | Muito alta |
| **PSPACE** | Espaço polinomial | TQBF | Potencialmente maior que NP |
| **L** | Logspace determinista | Verificação linear | Muito baixa |
| **NL** | Logspace não-determinista | Caminho em grafo | Baixa |

### 🧪 Desafios para ti

1. Dá um exemplo de um problema que está em NP, mas que não se sabe se está em P.
2. Tenta reduzir **3-SAT → CLIQUE** com um exemplo pequeno.
3. Constrói uma instância simples do **TQBF** e verifica se é verdadeira.
4. Analisa o jogo **Generalized Geography** num grafo com 4 vértices — quem vence?

## 🧬 **10. Hierarquia de Linguagens e Modelos de Computação**

Esta é uma das secções mais estruturantes da Teoria da Computação: compreender como diferentes **classes de linguagens** se posicionam numa **hierarquia de poder computacional**, e como **os autómatos associados** modelam esses níveis crescentes de complexidade. 🌐📈

### 🔹 10.1 Hierarquia de Chomsky:

Esta **hierarquia formal**, proposta nos anos 1950, **classifica linguagens formais** com base na complexidade das gramáticas que as geram:

| Tipo de Linguagem | Tipo de Gramática | Autómato Associado | Poder Computacional |
| --- | --- | --- | --- |
| **Regular (RL)** | Gramáticas regulares (lineares) | DFA/NFA | Baixo |
| **Livre de Contexto (CFL)** | CFG | PDA | Médio |
| **Sensível ao Contexto (CSL)** | CSG | LBA | Elevado |
| **Recursivamente Enumerável (RE)** | Gramáticas não-restritas | Turing Machine | Máximo |

### 🔹 10.2 Gramáticas Contextuais e Não-Restritas

#### 🟨 **Context-Sensitive Grammars (CSG)**

* As produções têm a forma:
* Ou seja, a substituição da variável **A** depende do **contexto** em que ocorre.

💡 Linguagens sensíveis ao contexto são **mais expressivas** que as CFL, mas ainda **decidíveis**.

#### 🟥 **Gramáticas Não-Restritas (Type 0)**

* Produções sem restrições:
* Podem gerar **todas as linguagens reconhecíveis por uma Máquina de Turing** (linguagens RE).

⚠️ Algumas linguagens aqui **não são decidíveis**!

### 🔹 10.3 LBA – Autómatos Linearmente Limitados

🧠 Um **Linear Bounded Automaton** (LBA) é uma **Máquina de Turing** com a fita **limitada ao tamanho da entrada** multiplicado por uma constante.

Características:

* Reconhece **linguagens sensíveis ao contexto (CSL)**
* Determina **se um string pertence a uma CSL** sem explorar fita infinita

💡 Essencial para demonstrar que existe **uma classe acima de CFLs** mas **ainda abaixo das RE** em termos de expressividade.

### 🔹 10.4 Comparação de Poderes Computacionais

A Hierarquia de Chomsky também nos ajuda a **comparar o poder dos modelos computacionais**:

| Modelo | Linguagens Aceites | Capacidade de Memória |
| --- | --- | --- |
| DFA/NFA | Regular | Estados finitos |
| PDA | Livre de Contexto (CFL) | Pilha |
| LBA | Sensível ao Contexto (CSL) | Memória linearmente limitada |
| Máquina de Turing | Recursivamente Enumerável | Fita infinita bidirecional |

🔁 **Quanto mais expressivo o modelo, mais complexo e menos decidível o problema se torna.**

### 📚 Exemplo visual da Hierarquia

+-------------------------------+  
| Recursively Enumerable (RE) |  
| ↳ Máquinas de Turing |  
| ↳ Gramáticas tipo 0 |  
| ↳ Pode não terminar |  
+---------------↑--------------+  
| Context-Sensitive Languages |  
| ↳ LBAs |  
| ↳ Gramáticas contextuais |  
+---------------↑--------------+  
| Context-Free Languages |  
| ↳ PDAs |  
| ↳ CFGs |  
+---------------↑--------------+  
| Regular Languages |  
| ↳ DFAs / NFAs |  
| ↳ Gramáticas Regulares |  
+-------------------------------+

### 🌟 Aplicações e Reflexões

* **Análise léxica**? Linguagens regulares são suficientes ✅
* **Parsing de linguagens de programação**? Usamos CFLs ✅
* **Sistemas que exigem mais contexto**? Pode ser necessário um LBA ou MT 🚀
* **Quer saber se o seu programa termina?** → Pode ser indecidível 😵

### 🧪 Desafios práticos

1. Constrói uma gramática sensível ao contexto para
2. Mostra que uma determinada linguagem não é CFL mas é CSL
3. Compara o comportamento de uma MT e um LBA sobre a mesma entrada
4. Faz uma tabela de produção para uma gramática não-restrita

Dedicado a: 1) Dino Mandrioli e Carlo Ghezzi, e 2) Derick Wood

Autores dos dois livros sobre este tema que me fascinaram na década de 90.  
  
[Uma imagem com símbolo, Tipo de letra, Gráficos, captura de ecrã

Os conteúdos gerados por IA poderão estar incorretos.](https://creativecommons.org/licenses/by-nc/4.0/)Luís Simões da Cunha (2025)