Uma imagem com texto, captura de ecrã, diagrama, desenho

Os conteúdos gerados por IA poderão estar incorretos.

Breve Tutorial, por Luís Simões da Cunha (2025)

[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/)

Este tutorial só foi possível devido à emergência da IA Generativa

Deixo aqui a minha homenagem e gratidão aos Professores

Dino Mandrioli e Carlo Ghezzi

Uma imagem com texto, livro, Retângulo, Impressão

Os conteúdos gerados por IA poderão estar incorretos.

e Derick Wood

Uma imagem com texto, livro, Tipo de letra, Retângulo

Os conteúdos gerados por IA poderão estar incorretos.  
  
  
  
Por terem escrito duas das obras que mais me fascinaram nos anos 90.

**Convivi** com elas na maravilhosa biblioteca da Faculdade de Letras da UP ❤️

Índice

[🧭 Parte I – Fundamentos e Motivação 10](#_Toc195918712)

[📍 **1. O que é a Teoria da Computação?** 10](#_Toc195918713)

[📖 1.1 A História e a Importância da Teoria da Computação 10](#_Toc195918714)

[🌍 1.2 Aplicações Práticas na Informática 10](#_Toc195918715)

[🧭 1.3 Como Está Organizado Este Tutorial 11](#_Toc195918716)

[✍️ Para Reflectir… 11](#_Toc195918717)

[🧭 Parte I – Fundamentos e Motivação 12](#_Toc195918718)

[🧠 **2. Pré-requisitos Matemáticos Essenciais** 12](#_Toc195918719)

[🔢 2.1 Conjuntos, Relações e Funções 12](#_Toc195918720)

[🌲 2.2 Grafos e Árvores 13](#_Toc195918721)

[🧩 2.3 Provas: Construção, Contradição e Indução 13](#_Toc195918722)

[🔤 2.4 Strings, Linguagens e Operações 14](#_Toc195918723)

[🧪 Experimenta tu mesmo/a! 14](#_Toc195918724)

[📌 Resumo Visual 14](#_Toc195918725)

[🧩 Parte II – Linguagens Regulares e Autómatos Finitos 16](#_Toc195918726)

[🤖 **3. Autómatos Finitos Deterministas (DFA)** 16](#_Toc195918727)

[🎯 3.1 Conceito Intuitivo 16](#_Toc195918728)

[🧱 3.2 Notação Formal 16](#_Toc195918729)

[🧮 3.3 Exemplo Visual 16](#_Toc195918730)

[🎨 3.4 Diagramas de Transição 17](#_Toc195918731)

[⚙️ 3.5 Linguagem Aceite por um DFA 17](#_Toc195918732)

[🧠 3.6 Construção de DFAs 18](#_Toc195918733)

[🧪 Experimenta tu mesmo/a! 18](#_Toc195918734)

[📌 Resumo Visual 18](#_Toc195918735)

[🧩 Parte II – Linguagens Regulares e Autómatos Finitos 19](#_Toc195918736)

[🌀 **4. Autómatos Finitos Não Deterministas (NFA e NFA-ε)** 19](#_Toc195918737)

[🤔 4.1 Diferenças e Intuição 19](#_Toc195918738)

[🧮 4.2 Definição Formal de um NFA 20](#_Toc195918739)

[🎨 4.3 Exemplo Visual (NFA simples) 20](#_Toc195918740)

[🔁 4.4 NFA com Transições ε (NFA-ε) 20](#_Toc195918741)

[🔄 4.5 Conversão de NFA para DFA 21](#_Toc195918742)

[⚙️ 4.6 Autómatos com Saídas: Mealy e Moore 21](#_Toc195918743)

[🧪 Experimenta tu mesmo/a! 21](#_Toc195918744)

[📌 Resumo Visual 22](#_Toc195918745)

[🧩 Parte II – Linguagens Regulares e Autómatos Finitos 22](#_Toc195918746)

[✍️ **5. Expressões Regulares e Gramáticas Regulares** 22](#_Toc195918747)

[🧠 5.1 O que é uma Expressão Regular? 23](#_Toc195918748)

[🧮 5.2 Construção de Expressões Regulares 23](#_Toc195918749)

[💬 Exemplos 23](#_Toc195918750)

[🔁 5.3 De ER para Autómato (e vice-versa) 23](#_Toc195918751)

[📜 5.4 Gramáticas Regulares 24](#_Toc195918752)

[💬 Exemplo de Gramática Regular 24](#_Toc195918753)

[🧪 Experimenta tu mesmo/a! 24](#_Toc195918754)

[🔄 5.5 Equivalência entre Representações 25](#_Toc195918755)

[📌 Resumo Visual 25](#_Toc195918756)

[🧩 Parte II – Linguagens Regulares e Autómatos Finitos 26](#_Toc195918757)

[🧪 **6. Propriedades das Linguagens Regulares** 26](#_Toc195918758)

[🔁 6.1 Propriedades de Fecho 26](#_Toc195918759)

[🔍 6.2 Decidibilidade: o que conseguimos sempre saber 26](#_Toc195918760)

[💣 6.3 O Lema do Bombeamento (Pumping Lemma) 27](#_Toc195918761)

[⚔️ Usar o Lema para provar *não-regularidade* 27](#_Toc195918762)

[💬 Exemplo Clássico 28](#_Toc195918763)

[🧪 Experimenta tu mesmo/a! 28](#_Toc195918764)

[📌 Resumo Visual 28](#_Toc195918765)

[🧱 Parte III – Linguagens Livre de Contexto e Autómatos com Pilha 29](#_Toc195918766)

[📘 **7. Gramáticas Livre de Contexto (CFG)** 29](#_Toc195918767)

[✨ 7.1 O que é uma CFG? 29](#_Toc195918768)

[🌱 7.2 Derivações: como nascem as frases 30](#_Toc195918769)

[🌳 7.3 Árvores de Derivação 30](#_Toc195918770)

[⚠️ 7.4 Ambiguidade, Recursividade e Factoring 30](#_Toc195918771)

[🧪 7.5 Exemplos de CFGs clássicas 31](#_Toc195918772)

[🔧 7.6 Notações úteis: BNF 31](#_Toc195918773)

[🔍 7.7 Comparação com outras linguagens 31](#_Toc195918774)

[🧠 Dica pedagógica 31](#_Toc195918775)

[📌 Resumo Visual 32](#_Toc195918776)

[🧱 Parte III – Linguagens Livre de Contexto e Autómatos com Pilha 32](#_Toc195918777)

[🧰 **8. Autómatos com Pilha (PDA)** 32](#_Toc195918778)

[🤖 8.1 O que é um Autómato com Pilha? 33](#_Toc195918779)

[🧮 8.2 Definição Formal de um PDA 33](#_Toc195918780)

[🔁 8.3 Funcionamento Geral 33](#_Toc195918781)

[📘 8.4 Exemplo: Linguagem 34](#_Toc195918782)

[🧠 8.5 PDA Determinístico vs Não-Determinístico 34](#_Toc195918783)

[🔄 8.6 Equivalência com CFGs 34](#_Toc195918784)

[🧪 Experimenta tu mesmo/a! 34](#_Toc195918785)

[📌 Resumo Visual 35](#_Toc195918786)

[🧱 Parte III – Linguagens Livre de Contexto e Autómatos com Pilha 35](#_Toc195918787)

[📐 **9. Propriedades das Linguagens Livre de Contexto** 35](#_Toc195918788)

[🔁 9.1 Fecho das Linguagens Livre de Contexto 36](#_Toc195918789)

[✅ 9.2 Problemas Decidíveis para CFLs 36](#_Toc195918790)

[💣 9.3 Lema do Bombeamento para CFLs 36](#_Toc195918791)

[⚔️ Como usar o lema: 37](#_Toc195918792)

[🧪 Exemplo prático 37](#_Toc195918793)

[🔬 9.4 O Lema de Ogden (refinamento do bombeamento) 37](#_Toc195918794)

[📘 9.5 Linguagens que NÃO são CFLs 38](#_Toc195918795)

[📌 Resumo Visual 38](#_Toc195918796)

[🧠 Parte IV – Turing, Computabilidade e Modelos Avançados 39](#_Toc195918797)

[🖥️ **10. Máquinas de Turing** 39](#_Toc195918798)

[🤔 10.1 O que é uma Máquina de Turing? 39](#_Toc195918799)

[🧮 10.2 Definição Formal 39](#_Toc195918800)

[📋 10.3 Como funciona? 40](#_Toc195918801)

[🔄 10.4 A Máquina como Aceitadora e como Transdutor 40](#_Toc195918802)

[📊 10.5 Representação: Diagrama de Estados 40](#_Toc195918803)

[🔎 10.6 Descrição Instantânea (ID) 40](#_Toc195918804)

[🧠 10.7 Exemplo prático 41](#_Toc195918805)

[⚖️ 10.8 Comparação com outros modelos 41](#_Toc195918806)

[📌 Resumo Visual 41](#_Toc195918807)

[🧠 Parte IV – Turing, Computabilidade e Modelos Avançados 42](#_Toc195918808)

[🧬 **11. Modelos Variantes de Turing** 42](#_Toc195918809)

[🧯 11.1 Porquê estudar variantes? 42](#_Toc195918810)

[💽 11.2 Máquina de Turing com Múltiplas Fitas 43](#_Toc195918811)

[🌌 11.3 Máquina de Turing Não-Determinista 43](#_Toc195918812)

[🌐 11.4 Máquina de Turing Universal (MTU) 43](#_Toc195918813)

[🚫 11.5 Máquina com Fita Infinita em Ambas as Direções 44](#_Toc195918814)

[💡 11.6 Outras variantes curiosas 44](#_Toc195918815)

[📌 Resumo Visual 44](#_Toc195918816)

[🧪 Experimenta tu mesmo/a! 44](#_Toc195918817)

[🧠 Conclusão 44](#_Toc195918818)

[🧠 Parte IV – Turing, Computabilidade e Modelos Avançados 45](#_Toc195918819)

[⚖️ **12. Decidibilidade e Indecidibilidade** 45](#_Toc195918820)

[✅ 12.1 Problemas Decidíveis 45](#_Toc195918821)

[🚫 12.2 Problemas Indecidíveis 46](#_Toc195918822)

[💡 12.3 A Prova por Diagonalização 46](#_Toc195918823)

[🤯 12.4 Outros Exemplos de Problemas Indecidíveis 46](#_Toc195918824)

[📜 12.5 A Correspondência de Post 47](#_Toc195918825)

[📌 Resumo Visual 47](#_Toc195918826)

[🧪 Experimenta tu mesmo/a! 47](#_Toc195918827)

[🧠 Parte IV – Turing, Computabilidade e Modelos Avançados 48](#_Toc195918828)

[🧩 **13. Funções Recursivas e o Lado Matemático da Computação** 48](#_Toc195918829)

[🧠 13.1 O que é uma função computável? 49](#_Toc195918830)

[⚙️ 13.2 Funções Primitivas Recursivas 49](#_Toc195918831)

[🔁 13.3 Funções μ-Recursivas (ou Generalizadas) 49](#_Toc195918832)

[🧨 13.4 A Função de Ackermann 50](#_Toc195918833)

[📏 13.5 Gödelização e Números de Gödel 50](#_Toc195918834)

[🧮 13.6 O Teorema de Rice 50](#_Toc195918835)

[📐 13.7 A Diagonalização de Cantor 50](#_Toc195918836)

[🧠 13.8 λ-Cálculo e Computação 50](#_Toc195918837)

[📌 Resumo Visual 51](#_Toc195918838)

[🧪 Experimenta tu mesmo/a! 51](#_Toc195918839)

[🧠 Parte IV – Turing, Computabilidade e Modelos Avançados 52](#_Toc195918840)

[🧩 **13. Funções Recursivas e o Lado Matemático da Computação** 52](#_Toc195918841)

[📍 13.1 O Conceito de Função Computável 52](#_Toc195918842)

[🧱 13.2 Funções Primitivas Recursivas 52](#_Toc195918843)

[⚠️ 13.3 Limitações: Precisamos de mais... 53](#_Toc195918844)

[🔄 13.4 Funções μ-Recursivas 53](#_Toc195918845)

[📈 13.5 A Função de Ackermann 53](#_Toc195918846)

[🔢 13.6 Números de Gödel (Gödelização) 53](#_Toc195918847)

[🧪 13.7 O Teorema de Rice 54](#_Toc195918848)

[🔁 13.8 A Diagonalização de Cantor 54](#_Toc195918849)

[🧩 13.9 λ-Cálculo: O modelo funcional da computação 54](#_Toc195918850)

[📌 Resumo Tablado 54](#_Toc195918851)

[🧪 Desafios para ti! 55](#_Toc195918852)

[🔍 **O que é a Diagonalização?** 56](#_Toc195918853)

[👨‍🏫 Analogia intuitiva: a lista infinita de combinações 56](#_Toc195918854)

[🧠 Aplicação à Teoria da Computação 56](#_Toc195918855)

[🎲 Passo 1: Representar máquinas como strings 56](#_Toc195918856)

[🧨 Passo 2: Construir uma linguagem diagonal 57](#_Toc195918857)

[📚 Interpretação Profunda 57](#_Toc195918858)

[📌 Resumo Visual 58](#_Toc195918859)

[🧪 Para experimentar: 58](#_Toc195918860)

[🚀 Parte V – Complexidade Computacional 58](#_Toc195918861)

[🧩 **14. Complexidade de Tempo e Espaço** 58](#_Toc195918862)

[⏱️ 14.1 Noções Fundamentais 59](#_Toc195918863)

[🔢 14.2 Notações Assintóticas 59](#_Toc195918864)

[🧮 14.3 Classes de Complexidade 59](#_Toc195918865)

[🧠 14.4 Problemas NP-Completos 60](#_Toc195918866)

[🔄 14.5 Reduções Polinomiais 60](#_Toc195918867)

[📘 14.6 Teorema de Cook–Levin 60](#_Toc195918868)

[💾 14.7 Complexidade de Espaço 61](#_Toc195918869)

[📌 Tabela Resumo 61](#_Toc195918870)

[🧪 Desafios para ti: 61](#_Toc195918871)

[🔧 Parte VI – Aplicações e Temas Avançados 62](#_Toc195918872)

[🧵 **15. Gramáticas e Compiladores** 62](#_Toc195918873)

[📍 15.1 O que é um compilador? 62](#_Toc195918874)

[🔠 15.2 Análise Léxica (Scanner) 63](#_Toc195918875)

[🌳 15.3 Análise Sintática (Parser) 63](#_Toc195918876)

[🔄 15.4 Parsing Top-Down vs Bottom-Up 63](#_Toc195918877)

[⚙️ 15.5 Algoritmos Famosos 64](#_Toc195918878)

[🛠️ 15.6 Ferramentas do mundo real 64](#_Toc195918879)

[🎓 15.7 Do formalismo à prática 64](#_Toc195918880)

[📌 Resumo Visual 65](#_Toc195918881)

[🧪 Desafios para ti: 65](#_Toc195918882)

[🔧 Lex/Flex e Yacc/Bison – Onde a Teoria Ganha Vida! 65](#_Toc195918883)

[🧩 1. Lex e Flex – O Scanner Gerador 65](#_Toc195918884)

[🧠 2. Yacc e Bison – O Mestre da Análise Sintática 66](#_Toc195918885)

[🔗 Como se ligam Lex e Yacc? 66](#_Toc195918886)

[🧪 Um mini-exemplo funcional 67](#_Toc195918887)

[🔬 Relação com a Teoria da Computação 67](#_Toc195918888)

[🤓 Curiosidade histórica 68](#_Toc195918889)

[📚 Sugestão para o teu tutorial 68](#_Toc195918890)

[🔧 Parte VI – Aplicações e Temas Avançados 69](#_Toc195918891)

[🌌 **16. Temas Avançados e Exploratórios** 69](#_Toc195918892)

[🧪 16.1 Autómatos Probabilísticos (PFA) 69](#_Toc195918893)

[🧬 16.2 Autómatos Celulares 69](#_Toc195918894)

[🧮 16.3 Gramáticas Matriciais 70](#_Toc195918895)

[🌿 16.4 L-Systems (Sistemas de Lindenmayer) 70](#_Toc195918896)

[📊 Comparação geral dos modelos 71](#_Toc195918897)

[🧪 Desafios para ti 71](#_Toc195918898)

[📌 Resumo da secção 71](#_Toc195918899)

Espaço intencionalmente deixado vazio

## 🧭 Parte I – Fundamentos e Motivação

# 📍 **1. O que é a Teoria da Computação?**

Um mergulho fascinante no coração da ciência da computação 🧠✨

### 📖 1.1 A História e a Importância da Teoria da Computação

Já imaginaste como conseguimos saber **o que uma máquina pode ou não fazer**, mesmo antes de a construirmos? 🤔

É aqui que entra a **Teoria da Computação** — um ramo fundamental da ciência da computação que estuda **os limites do que é computável**. Vai muito além da programação: ajuda-nos a perceber **quais os problemas que conseguimos resolver com algoritmos** e **quais os que nunca conseguiremos resolver, independentemente do poder computacional disponível**. 🚫🖥️

A sua origem remonta a nomes como:

* **Alan Turing**, com a sua mítica “Máquina de Turing” (1936), que define o conceito formal de algoritmo;
* **Alonzo Church**, com o cálculo lambda;
* **Kurt Gödel**, com os teoremas da incompletude (que chocaram o mundo matemático).

Estes pioneiros demonstraram que **há problemas cuja resposta nunca poderá ser conhecida por nenhum computador**, por mais potente que seja.

### 🌍 1.2 Aplicações Práticas na Informática

Pode parecer teórico… mas a Teoria da Computação está por todo o lado! Eis alguns exemplos reais:

| Domínio | Aplicação Prática |
| --- | --- |
| 🔤 **Compiladores** | Conversão de código-fonte para código executável baseia-se em **gramáticas formais** e **autómatos** |
| 🔎 **Pesquisa em bases de dados** | Uso de **expressões regulares** e **autómatos finitos** para filtrar e analisar texto |
| 🔒 **Segurança** | Verificação de protocolos com base em modelos formais |
| ⚙️ **Inteligência Artificial** | Análise da **decidibilidade** de certos problemas |
| 📦 **Compressão de dados** | Baseada em padrões detetados por autómatos e linguagens regulares |

### 🧭 1.3 Como Está Organizado Este Tutorial

Para facilitar a tua aprendizagem e reduzir a sobrecarga cognitiva, organizámos este tutorial em **partes progressivas**, com uma abordagem do **simples ao complexo**, sempre com muitos exemplos, analogias, e exercícios interativos.

Cada capítulo segue esta estrutura:

1. 🎯 **Motivação clara** – Para que serve aprender isto?
2. 📦 **Conceitos fundamentais** – Com explicações visuais e linguagem acessível
3. 🛠️ **Exemplos práticos** – Com diagramas e resolução passo a passo
4. 🧪 **Experimenta tu mesmo/a** – Pequenos desafios para consolidar
5. 📚 **Resumo visual** – O essencial em poucos minutos

### ✍️ Para Reflectir…

* O que significa “resolver” um problema com um algoritmo?
* Haverá problemas que **nenhum computador, nunca**, conseguirá resolver?
* Estarás preparado para descobrir os limites do que é **possível computar**?

Bem-vindo à aventura mais filosófica e lógica da computação. O que vais aprender aqui **vai mudar para sempre a forma como pensas sobre algoritmos, máquinas e o próprio conhecimento**. 🚀

## 🧭 Parte I – Fundamentos e Motivação

# 

# 🧠 **2. Pré-requisitos Matemáticos Essenciais**

Tudo o que precisas de saber antes de mergulhar nos autómatos, gramáticas e máquinas de Turing 🧩🧮

Antes de começares a construir **autómatos**, escrever **gramáticas** ou entender **o que é computável**, precisas de dominar alguns conceitos matemáticos fundamentais. Não te preocupes — vamos rever tudo de forma intuitiva e com exemplos bem simples.

### 🔢 2.1 Conjuntos, Relações e Funções

#### 📦 Conjuntos

Um **conjunto** é uma coleção de elementos. Pode ser finito ou infinito.

Exemplo:

Operações principais:

* União:
* Interseção:
* Diferença:
* Produto cartesiano:

#### 🔗 Relações

Uma **relação** é qualquer conjunto de pares ordenados.

Exemplo:

Relações podem ter propriedades como **reflexividade**, **simetria**, **transitividade**.

#### 🎯 Funções

Uma **função** associa *cada elemento de um conjunto* a *um único elemento de outro*.

Exemplo:

Tipos comuns:

* Injetiva (sem repetições no contradomínio)
* Sobrejetiva (preenche o contradomínio)
* Bijetiva (é ambos!)

### 🌲 2.2 Grafos e Árvores

#### 🌐 Grafos

Um **grafo** é um conjunto de **vértices** (nós) ligados por **arestas** (ligações).

Usamos grafos para representar:

* Autómatos finitos
* Caminhos de computação
* Redes de computadores

Exemplo:

A — B — C  
 \ /  
 D — E

#### 🌳 Árvores

Uma **árvore** é um tipo especial de grafo **sem ciclos**, com um **nó raiz** e **ramos**.  
Importante para representar **derivações de gramáticas** (parse trees).

### 🧩 2.3 Provas: Construção, Contradição e Indução

Vamos usar **provas matemáticas** para mostrar que:

* Autómatos funcionam como esperado
* Linguagens não pertencem a certas classes
* Certas propriedades são verdadeiras em geral

Três tipos comuns:

1. **Por construção** – “Aqui está um exemplo que funciona.”
2. **Por contradição** – “Suponhamos que sim... mas dá erro!”
3. **Por indução** – Provas passo a passo para todas as entradas (como dominós!).

Exemplo de indução (soma dos n primeiros inteiros):

### 🔤 2.4 Strings, Linguagens e Operações

#### 🧱 Strings

Uma **string** é uma sequência de símbolos.

Exemplo:  
Para um alfabeto , podemos ter:

ε (string vazia), a, ab, bba, ...

#### 📚 Linguagens

Uma **linguagem** é um conjunto de strings.

Exemplo:

Operações com linguagens:

* União:
* Concatenação:
* Estrela de Kleene: – todas as repetições possíveis

### 🧪 Experimenta tu mesmo/a!

1. Se e , qual é o produto cartesiano ?
2. Qual destas strings pertence a ?  
   a) "abba"  
   b) "abc"  
   c) "" (vazia)
3. Que tipo de prova é esta? "Assumimos que uma linguagem é regular... mas dá uma contradição."

### 📌 Resumo Visual

| Conceito | Explicação Rápida | Exemplo |
| --- | --- | --- |
| Conjunto | Coleção de elementos |  |
| Função | Cada input → 1 output |  |
| Grafo | Vértices + arestas | Diagrama de autómatos |
| Árvore | Grafo sem ciclos | Árvore de derivação |
| String | Sequência de símbolos | "aab" |
| Linguagem | Conjunto de strings |  |

## 🧩 Parte II – Linguagens Regulares e Autómatos Finitos

# 🤖 **3. Autómatos Finitos Deterministas (DFA)**

Máquinas simples que sabem dizer “sim” ou “não” a padrões… com precisão matemática!

### 🎯 3.1 Conceito Intuitivo

Um **autómato finito determinista** (DFA) é como um *guardião de portagem* com regras claras:

* Lê um símbolo de cada vez;
* Segue um único caminho;
* No fim, decide se aceita ou rejeita uma palavra.

É uma **máquina teórica** que lê strings de um **alfabeto finito** e decide se pertencem ou não a uma **linguagem regular**.

Imagina um DFA como um pequeno robô que lê uma palavra letra a letra, mudando de “sala” (estado) de acordo com o que lê.

### 🧱 3.2 Notação Formal

Um DFA é um quíntuplo:

🔍 Onde:

* : conjunto finito de **estados**
* : alfabeto (símbolos possíveis de entrada)
* : função de transição ()
* : estado **inicial**
* : conjunto de **estados finais** (ou de aceitação)

### 🧮 3.3 Exemplo Visual

#### Objetivo:

Aceitar todas as strings sobre com número par de letras 'a'.

Estado: Lê: Vai para:  
------ ----- ---------  
 q0 a q1  
 q0 b q0  
 q1 a q0  
 q1 b q1

🎯 Estado inicial: q0  
✅ Estado final: q0 (porque número par de 'a')

💡 Interpretação:

* q0: número par de 'a'
* q1: número ímpar de 'a'

### 🎨 3.4 Diagramas de Transição

Representamos DFAs com **nós (estados)** e **setas (transições)**:

b a  
 ┌───► q1 ───►┐  
 │ │  
q0◄─────────────┘  
 ▲ b  
 │  
 ▼  
[ q0 ] ← estado inicial e final

* Círculo duplo = estado final ✅
* Seta de entrada = estado inicial 🟢
* Etiquetas = símbolos lidos

### ⚙️ 3.5 Linguagem Aceite por um DFA

Dado um DFA , dizemos que **aceita uma palavra** se a leitura de todos os símbolos de , a partir de , termina num estado de .

💬 Exemplo:  
Com o DFA acima, a string “abba” é aceite?

Passo a passo:

q0 -a→ q1  
q1 -b→ q1  
q1 -b→ q1  
q1 -a→ q0 ✅

✅ Termina em q0 → **Aceite!**

### 🧠 3.6 Construção de DFAs

Para construir um DFA, pensa como se fosses o robô:

1. Quais são os **requisitos da linguagem**?
2. Que **estados** preciso para guardar “memória”?
3. Como **mudo de estado** ao ler cada símbolo?
4. Quais são os estados finais?

💡Dica: Pensa em *“modos de pensar”* — cada estado representa um *modo de saber o que leste até agora*.

### 🧪 Experimenta tu mesmo/a!

Cria um DFA que aceite apenas strings de 0s e 1s que terminem em **"01"**.

👉 Dica:

* Estado inicial: q0
* Precisas de “lembrar” o que foi lido anteriormente!
* Faz os estados “memorizar” os últimos símbolos!

### 📌 Resumo Visual

| Componente | Significado |
| --- | --- |
|  | Estados |
|  | Alfabeto |
|  | Função de transição |
|  | Estado inicial |
|  | Estados de aceitação |
| Diagrama | Representação visual do comportamento |
| Aceitação | Palavras que terminam num estado de |

## 🧩 Parte II – Linguagens Regulares e Autómatos Finitos

# 🌀 **4. Autómatos Finitos Não Deterministas (NFA e NFA-ε)**

E se uma máquina pudesse explorar várias possibilidades ao mesmo tempo?

### 🤔 4.1 Diferenças e Intuição

Ao contrário dos DFAs, onde **cada símbolo leva a um único estado**, nos NFAs:

* Podemos ter **múltiplas transições** com o mesmo símbolo;
* Ou **nenhuma transição**;
* Ou mesmo **transições “invisíveis”**, chamadas **ε-transições**, que permitem saltar de estado **sem ler nenhum símbolo**!

💡 Imagina um NFA como um **explorador de caminhos**: a cada letra lida, ele **testa todas as possibilidades ao mesmo tempo** — como se fosse um super-herói com o poder da multiplicação! 🦸‍♂️✨

### 🧮 4.2 Definição Formal de um NFA

Um NFA é um quíntuplo:

Onde:

* : conjunto de estados
* : alfabeto de entrada
* : função de transição que **pode devolver vários estados!**
* : estado inicial
* : estados finais

➡️ A diferença fundamental está na função :  
Em vez de dar um único estado, ela dá um **conjunto de estados possíveis**!

### 🎨 4.3 Exemplo Visual (NFA simples)

Aceita todas as strings que **terminam em "01"** sobre .

0 1  
q0 ───► q1 ───► q2  
 ↘  
 1  
 ↘  
 q0

* Estado inicial: q0
* Estado final: q2
* De q0, ao ler 0, pode ir para q1
* De q0, ao ler 1, pode continuar em q0 (loop)
* De q1, ao ler 1, vai para q2 ✅

✅ Se alguma sequência terminar em "01", pelo menos um dos caminhos **leva a q2** → a string é aceite.

### 🔁 4.4 NFA com Transições ε (NFA-ε)

Às vezes, um NFA pode **mudar de estado sem ler símbolo nenhum** — como se tivesse **atalhos secretos**. Chamamos isso de **transição ε**.

🔄 A função de transição agora é:

💡 Isto permite criar “caminhos de preparação” antes da leitura real — como se a máquina aquecesse antes da corrida 🏃‍♂️🔥

### 🔄 4.5 Conversão de NFA para DFA

**Notícia incrível:** apesar de parecerem mais poderosos, **todos os NFAs têm um DFA equivalente**! 😲

Este processo chama-se **construção do autómato de subconjuntos (powerset construction)**:

1. Cada estado do novo DFA representa um **conjunto de estados do NFA**.
2. Para cada símbolo, vemos **para onde todos esses estados iriam no NFA**.
3. Repetimos até cobrir todas as combinações.

🎯 Pode dar origem a muitos estados, mas garante que o novo DFA reconhece **exatamente a mesma linguagem**.

### ⚙️ 4.6 Autómatos com Saídas: Mealy e Moore

Estes são **autómatos que emitem saídas**, não só aceitam/rejeitam:

* **Mealy Machine**: saída depende do **estado atual + símbolo lido**
* **Moore Machine**: saída depende **apenas do estado atual**

Exemplo simples:

| Estado | Entrada | Novo estado | Saída |
| --- | --- | --- | --- |
| q0 | 1 | q1 | 0 |
| q1 | 0 | q0 | 1 |

Usados em aplicações práticas como:

* **Codificadores/descodificadores**
* **Controladores lógicos**
* **Protocolos de comunicação**

### 🧪 Experimenta tu mesmo/a!

🧠 Constrói um NFA para a linguagem

Dica:

* Estado q0: início
* Estado q1: já viu um “a”
* Estado q2: viu “ab” → aceitar!

Inclui uma transição ε se quiseres aceitar palavras mais curtas por “atalho”!

### 📌 Resumo Visual

| Tipo de Autómato | Transições | Caminhos possíveis | Saídas |
| --- | --- | --- | --- |
| DFA | 1 por símbolo | 1 único | Não |
| NFA | Várias por símbolo | Múltiplos caminhos | Não |
| NFA-ε | Pode mudar sem símbolo | + caminhos ainda | Não |
| Mealy/Moore | Com saída | Determinísticos | Sim |

## 🧩 Parte II – Linguagens Regulares e Autómatos Finitos

# ✍️ **5. Expressões Regulares e Gramáticas Regulares**

O poder de escrever linguagens com símbolos simples — e transformar texto em estrutura!

### 🧠 5.1 O que é uma Expressão Regular?

Uma **expressão regular** (ER) é uma fórmula que descreve **um padrão de strings** sobre um alfabeto. É como uma receita que diz: “aceita tudo o que começa com ‘a’, termina com ‘b’ e tem qualquer número de ‘c’ pelo meio”.

🔡 Alfabeto: conjunto de símbolos, ex:   
🎯 ERs descrevem **linguagens regulares**, ou seja, subconjuntos de

### 🧮 5.2 Construção de Expressões Regulares

As expressões regulares são construídas **recursivamente**:

1. **Casos base**:
   * – linguagem vazia
   * – string vazia
   * – símbolo individual
2. **Operações**:
   * **União**: (ou ) → um ou outro
   * **Concatenação**: → um seguido do outro
   * **Fecho de Kleene**: → zero ou mais repetições

### 💬 Exemplos

| Expressão Regular | Linguagem |
| --- | --- |
|  | todas as strings com qualquer número de 'a' |
|  | apenas "a" ou "b" |
|  | "", "ab", "abab", ... |
|  | qualquer combinação de "a" e "b" |
|  | qualquer número de "a", seguido de um "b" |

### 🔁 5.3 De ER para Autómato (e vice-versa)

#### ✔️ ER → NFA

* Para cada expressão regular, é possível construir **um NFA equivalente**.
* Existem **algoritmos sistemáticos** que fazem esta conversão.

#### ✔️ NFA → ER

* Também é possível fazer o caminho inverso: transformar qualquer DFA ou NFA numa expressão regular.
* Técnicas comuns: **eliminação de estados** ou **métodos matriciais (generalizações de Floyd-Warshall)**.

💡 Isto mostra que **ERs, NFAs e DFAs são equivalentes em poder expressivo** — descrevem exatamente as **linguagens regulares**.

### 📜 5.4 Gramáticas Regulares

Uma **gramática regular** é uma forma alternativa de descrever linguagens regulares, usando **regras de produção**:

* Forma geral:

🧩 São chamadas **gramáticas lineares à direita** (ou à esquerda) e geram as mesmas linguagens que os DFAs e ERs.

### 💬 Exemplo de Gramática Regular

S → aA   
A → bB   
B → ε

Gera apenas a string "ab".

📜 Podemos também ter:

S → aS | bS | ε

→ Gera todas as combinações de "a" e "b" (equivalente a )

### 🧪 Experimenta tu mesmo/a!

1. Constrói uma ER que aceite:
   * Strings que terminam em "ab"
   * Strings que têm **exatamente dois ‘a’**
2. Define uma gramática regular que aceite apenas as palavras: "a", "ab", "abb", "abbb"...

### 🔄 5.5 Equivalência entre Representações

| Representação | Equivalente a... |
| --- | --- |
| ER | NFA/DFA |
| NFA | ER |
| Gramática Reg. | ER / NFA / DFA |

👉 Tudo o que podes dizer com um, podes dizer com os outros. Só muda a forma de expressar — como dizer “olá” em várias línguas.

### 📌 Resumo Visual

| Símbolo | Significado |
| --- | --- |
| ou | União (ou) |
| ou nada | Concatenação |
|  | Fecho de Kleene |
|  | Linguagem vazia |
|  | String vazia |

## 🧩 Parte II – Linguagens Regulares e Autómatos Finitos

# 🧪 **6. Propriedades das Linguagens Regulares**

O que as linguagens regulares podem (e não podem) fazer — e como provar isso com elegância matemática!

### 🔁 6.1 Propriedades de Fecho

As linguagens regulares têm um comportamento muito previsível: estão **fechadas** sob várias operações.

Isto significa que, **se aplicarmos essas operações a linguagens regulares, o resultado continua a ser uma linguagem regular!** 😄

| Operação | Exemplo | Resultado é regular? |
| --- | --- | --- |
| União |  | ✅ |
| Concatenação |  | ✅ |
| Fecho de Kleene |  | ✅ |
| Complemento |  | ✅ |
| Interseção |  | ✅ |
| Diferença |  | ✅ |
| Reverso |  | ✅ |

📌 Isto permite-nos **criar novas linguagens a partir de antigas** e ainda assim manter a regularidade!

### 🔍 6.2 Decidibilidade: o que conseguimos sempre saber

Para linguagens regulares, conseguimos **responder de forma automática (algorítmica)** a muitas perguntas fundamentais:

| Problema | Podemos decidir? |
| --- | --- |
| Uma string pertence a ? | ✅ Sim (basta correr o DFA) |
| A linguagem é vazia? | ✅ Sim (ver se há caminho até estado final) |
| A linguagem é finita? | ✅ Sim |
| Dois DFAs aceitam a mesma linguagem? | ✅ Sim (problema da equivalência de DFAs) |

💡 Isto mostra o quão **bem-comportadas** são estas linguagens — e porque são tão usadas em áreas como **análise léxica, segurança, redes, e validação de padrões**.

### 💣 6.3 O Lema do Bombeamento (Pumping Lemma)

Agora a parte mais emocionante: e **como mostramos que uma linguagem *não* é regular?**

Usamos o **Lema do Bombeamento**, uma ferramenta clássica que mostra que certas linguagens **não podem ser descritas por nenhum DFA**.

#### 📜 O que diz o lema?

Se é uma linguagem regular, então **existe um número (comprimento de bombeamento)** tal que, para qualquer string com , podemos escrever:

Tal que:

* Para todo ,

👉 Em termos simples: **a parte “y” pode ser repetida (ou omitida) quantas vezes quisermos, e a string ainda pertence à linguagem.**

### ⚔️ Usar o Lema para provar *não-regularidade*

Para mostrar que uma linguagem **não** é regular:

1. Supõe que ela é regular (por absurdo).
2. Aplica o lema:
3. Mostra que há um valor de tal que
4. Conclusão: contradição → a linguagem **não é regular**

### 💬 Exemplo Clássico

Seja:

Supõe que L é regular com p=3  
Escolhe → "aaabbb"  
Qualquer divisão , com , vai fazer com que "y" tenha apenas "a"’s.  
Se repetirmos "y", temos mais "a"’s do que "b"’s → a string **não pertence** a L.

➡️ **Logo, L não é regular!**

### 🧪 Experimenta tu mesmo/a!

1. Prova que não é regular (com o lema do bombeamento)
2. Será que é regular?

👉 Dica: Se não consegues criar um DFA que “lembre-se” de contar, o lema do bombeamento pode ser teu aliado! ⚖️

### 📌 Resumo Visual

| Conceito | Descrição |
| --- | --- |
| Fecho | Combinar linguagens e manter regularidade |
| Decidibilidade | Podemos saber respostas automaticamente |
| Lema do Bombeamento | Ferramenta para provar que **não é regular** |
| Estratégia | Supor que é regular, aplicar o lema, mostrar contradição |

## 🧱 Parte III – Linguagens Livre de Contexto e Autómatos com Pilha

# 📘 **7. Gramáticas Livre de Contexto (CFG)**

A linguagem dos compiladores, a estrutura das linguagens… e o primeiro contacto sério com hierarquias formais.

### ✨ 7.1 O que é uma CFG?

Uma **gramática livre de contexto (CFG)** é um conjunto de regras que descreve **como construir strings válidas** de uma linguagem.

É composta por um quadruplo:

* : símbolos variáveis (não-terminais)
* : símbolos terminais (alfabeto)
* : conjunto de **regras de produção**
* : símbolo inicial

Cada regra de produção tem a forma:

A → α

Onde:

* é um **não-terminal**
* é uma string de **terminais e/ou não-terminais**

### 🌱 7.2 Derivações: como nascem as frases

As derivações mostram **como produzir strings** a partir do símbolo inicial, aplicando regras.

* **Derivação à esquerda**: substituímos sempre o **primeiro** não-terminal
* **Derivação à direita**: substituímos sempre o **último** não-terminal

💬 Exemplo:

Se , então:

🎄 Estas derivações podem ser representadas graficamente através de **árvores de derivação**!

### 🌳 7.3 Árvores de Derivação

Uma **árvore de derivação** mostra **a estrutura hierárquica** de como uma string é gerada.

* A raiz é o símbolo inicial
* Cada nível mostra a aplicação de uma produção
* As folhas são os **símbolos terminais** que formam a string

👁️ Ajuda-nos a visualizar ambiguidades!

### ⚠️ 7.4 Ambiguidade, Recursividade e Factoring

#### ❓Ambiguidade

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

Exemplo clássico:

E → E + E | E \* E | id

➡️ A expressão id + id \* id pode ter **duas interpretações**! Isso é ambiguidade.

🔧 Soluções:

* **Factoring**: reestruturar a gramática para evitar ambiguidade
* **Impor precedência** e associatividade nas regras

#### ♻️ Recursividade

* **Recursividade à esquerda**:
* **Recursividade à direita**:

Às vezes é necessário **eliminar a recursividade à esquerda** para facilitar o parsing!

### 🧪 7.5 Exemplos de CFGs clássicas

1. Linguagem

S → aSb | ε

1. Palíndromos sobre {a, b}:

S → aSa | bSb | a | b | ε

1. Aritmética simples:

E → E + T | T   
T → T \* F | F   
F → (E) | id

### 🔧 7.6 Notações úteis: BNF

BNF (Backus-Naur Form) é uma forma compacta e elegante de escrever CFGs.

Exemplo:

<expr> ::= <expr> + <term> | <term>   
<term> ::= <term> \* <factor> | <factor>   
<factor> ::= ( <expr> ) | id

### 🔍 7.7 Comparação com outras linguagens

| Tipo de linguagem | Máquina associada | Exemplo clássico |
| --- | --- | --- |
| Regular | DFA/NFA | a\*b\* |
| Livre de Contexto | PDA | aⁿbⁿ |
| Sensível ao contexto | LBA | aⁿbⁿcⁿ |
| Recursiva geral | Máquina de Turing | Qualquer linguagem computável |

### 🧠 Dica pedagógica

Faz sempre o seguinte ao estudar CFGs:

* Desenha árvores!
* Experimenta várias derivações para a mesma string
* Testa se consegues gerar strings **inválidas** (ajuda a identificar ambiguidades)
* Lembra-te: uma CFG pode gerar **linguagens infinitas**

### 📌 Resumo Visual

| Conceito | Descrição |
| --- | --- |
| CFG | Regras para gerar strings |
| Derivação | Aplicar regras passo a passo |
| Árvore de Derivação | Representação gráfica |
| Ambiguidade | Várias árvores para a mesma string |
| BNF | Notação concisa para CFGs |
| Recursividade | Quando um símbolo gera-se a si mesmo |

## 

## 🧱 Parte III – Linguagens Livre de Contexto e Autómatos com Pilha

# 🧰 **8. Autómatos com Pilha (PDA)**

Máquinas com memória que reconhecem linguagens livre de contexto

### 🤖 8.1 O que é um Autómato com Pilha?

Um **PDA** é uma máquina semelhante a um NFA, mas com uma **estrutura extra chamada pilha** que lhe permite lembrar de certas informações.

💡 A pilha funciona como uma *torre de pratos*: só consegues colocar (push) e retirar (pop) pratos do topo. É **LIFO (Last In, First Out)**.

### 🧮 8.2 Definição Formal de um PDA

Um PDA é um 7-tuplo:

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

👉 A máquina lê um símbolo de entrada, *vê o topo da pilha*, e decide o próximo estado e como alterar a pilha.

### 🔁 8.3 Funcionamento Geral

Num PDA, cada transição faz três coisas:

1. **Lê** um símbolo (ou ε)
2. **Verifica** o topo da pilha
3. **Decide**:
   * Qual o próximo estado
   * O que **retirar** da pilha (pop)
   * O que **colocar** na pilha (push)

💡 A pilha permite “lembrar” o que foi lido antes — essencial para linguagens como , em que é preciso “contar” de forma aninhada.

### 📘 8.4 Exemplo: Linguagem

A ideia:

* **Empurrar um símbolo na pilha** para cada "a" lido
* **Retirar um símbolo da pilha** para cada "b"

Se no fim a pilha estiver como começou e todos os símbolos tiverem sido lidos… ✅ string aceite!

### 🧠 8.5 PDA Determinístico vs Não-Determinístico

* **NPDA (não-determinístico)**: pode escolher entre várias transições
* **DPDA (determinístico)**: só pode haver no máximo **uma** transição válida por situação

🔍 Algumas linguagens livres de contexto **só podem ser reconhecidas por NPDAs**, nunca por DPDAs!  
➡️ Exemplo: a linguagem dos palíndromos

### 🔄 8.6 Equivalência com CFGs

**Toda a linguagem gerada por uma CFG pode ser reconhecida por um PDA**, e vice-versa.

| Representação | Linguagem gerada |
| --- | --- |
| CFG | Livre de contexto |
| PDA | Livre de contexto |

🎯 Isto mostra que **gramáticas e autómatos com pilha têm o mesmo poder descritivo** — duas faces da mesma moeda.

### 🧪 Experimenta tu mesmo/a!

Constrói um PDA que aceite:

1. (pista: empurra "A" para cada "a", e retira "A" para cada "b")
2. (pista: usa não-determinismo para “adivinhar” o meio da string)

### 📌 Resumo Visual

| Componente | Função |
| --- | --- |
| Pilha | Memória auxiliar |
| Push | Coloca símbolo na pilha |
| Pop | Retira símbolo da pilha |
| ε-transição | Muda de estado sem ler entrada |
| NPDA | Pode ter múltiplas escolhas |
| DPDA | Comportamento único por situação |
| CFG ↔ PDA | Modelos equivalentes para CFLs |

## 🧱 Parte III – Linguagens Livre de Contexto e Autómatos com Pilha

# 📐 **9. Propriedades das Linguagens Livre de Contexto**

Quando os autómatos com pilha não são suficientes… e precisamos de provas mais refinadas.

### 🔁 9.1 Fecho das Linguagens Livre de Contexto

As CFLs (**Context-Free Languages**) não são tão bem-comportadas como as linguagens regulares. Estão **fechadas** apenas sob algumas operações.

| Operação | Resultado é CFL? |
| --- | --- |
| União | ✅ Sim |
| Concatenação | ✅ Sim |
| Estrela de Kleene | ✅ Sim |
| Interseção | ❌ Não (contraexemplo: ) |
| Complemento | ❌ Não |
| Diferença | ❌ Não (herdada da interseção) |

💡 Mesmo com estas limitações, muitas operações úteis mantêm-se dentro do universo CFL — o que é vital em **análise sintática** e **parsing**!

### ✅ 9.2 Problemas Decidíveis para CFLs

Embora as CFLs tenham limitações, muitos problemas são **decidíveis**, ou seja, **há algoritmos que dão sempre resposta**:

| Problema | Decidível? |
| --- | --- |
| (pertinência) | ✅ Sim |
|  | ✅ Sim |
|  | ❌ Não |
| Equivalência de duas CFGs | ❌ Não (indecidível!) |
| Finitude da linguagem | ✅ Sim |

➡️ Isto é especialmente útil para **compiladores**, onde precisamos de verificar se o código é válido ou gerar árvores de derivação.

### 💣 9.3 Lema do Bombeamento para CFLs

Assim como para linguagens regulares, há um **lema do bombeamento** para linguagens livre de contexto — mas com uma forma mais complexa.

#### 📜 Enunciado:

Se é uma CFL, então existe um inteiro tal que, para qualquer string , com , podemos escrever:

Tal que:

1. Para todo ,

👉 O truque está em repetir ou remover as partes "v" e "x" e ainda assim permanecer na linguagem.

### ⚔️ Como usar o lema:

1. Supõe que a linguagem é CFL.
2. Escolhe uma string suficientemente grande .
3. Mostra que **qualquer** decomposição de com as condições acima **falha** para algum .
4. Conclusão: contradição → L **não é CFL**.

### 🧪 Exemplo prático

Vamos provar que:

**não** é uma CFL.

1. Supõe que é CFL.
2. Escolhe .
3. Para qualquer decomposição , pelo menos duas secções (a's, b's, c's) são alteradas, e a proporção é destruída.  
   ➡️ Conclusão: L não é CFL.

### 🔬 9.4 O Lema de Ogden (refinamento do bombeamento)

O **Lema de Ogden** é uma versão **mais poderosa** do lema anterior. Permite escolher **posições marcadas** na string para reforçar o argumento de não-CFL.

💡 Muito útil para linguagens onde o Lema do Bombeamento clássico falha por ser demasiado fraco.

### 📘 9.5 Linguagens que NÃO são CFLs

| Linguagem | Motivo |
| --- | --- |
|  | Requer mais do que uma pilha |
|  | Precisa de duplicar a string |
|  | Não pode ser representada com uma única memória linear (pilha) |

👉 Estas linguagens exigem **mais poder computacional**, como autómatos com duas pilhas ou máquinas de Turing.

### 📌 Resumo Visual

| Conceito | Descrição |
| --- | --- |
| CFL | Linguagens geradas por CFGs |
| Fecho | Algumas operações mantêm CFLs, outras não |
| Decidibilidade | Alguns problemas têm resposta automática |
| Lema do Bombeamento | Provar que uma linguagem não é CFL |
| Lema de Ogden | Versão mais refinada e poderosa |

## 🧠 Parte IV – Turing, Computabilidade e Modelos Avançados

# 🖥️ **10. Máquinas de Turing**

A mente abstrata da computação — capaz de simular qualquer algoritmo

### 🤔 10.1 O que é uma Máquina de Turing?

A Máquina de Turing (MT) é um **modelo matemático** idealizado para representar qualquer cálculo que um computador possa realizar. Ela serve como a **base teórica para a noção de algoritmo**.

É constituída por:

* **Uma fita infinita** (dividida em células), onde a informação é lida/escrita
* **Uma cabeça de leitura/escrita** que se move para a esquerda ou direita
* **Uma unidade de controlo finita** (semelhante ao cérebro da máquina)
* **Um conjunto de estados** e uma **função de transição**

### 🧮 10.2 Definição Formal

Uma MT é um 7-tuplo:

* : conjunto de estados
* : alfabeto de entrada (não inclui o símbolo branco ␣)
* : alfabeto da fita, onde e ␣ ∈
* : função de transição
* : estado inicial
* : estado de aceitação
* : estado de rejeição (≠ )

### 📋 10.3 Como funciona?

1. A cabeça **lê** o símbolo na célula atual da fita.
2. A função **decide**:
   * O novo estado
   * O novo símbolo a escrever
   * A direção do movimento: esquerda (L) ou direita (R)
3. O processo repete-se até atingir o estado de aceitação ou rejeição.

💡 A MT é determinista **por definição** (veremos mais à frente a versão não-determinista).

### 🔄 10.4 A Máquina como Aceitadora e como Transdutor

#### 📥 Como aceitadora de linguagens

A máquina começa com uma string de entrada escrita na fita e aceita se alcançar .

#### 🔄 Como transdutor (ou máquina computadora)

A MT pode também **transformar** uma entrada numa saída — por exemplo, calcular a soma de dois números binários.

### 📊 10.5 Representação: Diagrama de Estados

Cada transição pode ser representada por um arco:

q1 ──(a → b, R)──▶ q2

Interpretação:

* Lê "a"
* Escreve "b"
* Move-se para a Direita
* Passa do estado para

### 🔎 10.6 Descrição Instantânea (ID)

Para representar o estado da MT num determinado momento, usamos:

α q β

Onde:

* : conteúdo à esquerda da cabeça
* : estado atual
* : símbolo sob a cabeça e à sua direita

💬 Exemplo:

1q0110

Significa que a cabeça está sobre o primeiro 0, e o estado atual é q.

### 🧠 10.7 Exemplo prático

Vamos definir uma MT que reconhece a linguagem .

**Ideia:** substituir as por X, procurar o b correspondente e marcá-lo com Y, depois repetir.

💡 A fita será algo como:

a a a b b b ␣ ␣ ␣ ...  
↓

A cabeça substitui os as e bs por marcas, e no final verifica se está tudo bem marcado.

### ⚖️ 10.8 Comparação com outros modelos

| Modelo | Memória | Linguagens aceites |
| --- | --- | --- |
| DFA / NFA | Nenhuma | Regulares |
| PDA | Pilha | Livre de contexto |
| Máquina de Turing | Fita ilimitada | Recursivas e além |

🎯 A Máquina de Turing **engloba todos os modelos anteriores** e vai além — é o **modelo universal de computação**.

### 📌 Resumo Visual

| Conceito | Descrição |
| --- | --- |
| MT | Máquina abstrata que simula qualquer algoritmo |
| Fita | Memória ilimitada de leitura/escrita |
| Cabeça | Lê e escreve símbolos |
| ID | Estado momentâneo da execução |
| Aceitação | Chegada a |
| Transdução | Cálculo ou transformação de strings |

## 🧠 Parte IV – Turing, Computabilidade e Modelos Avançados

# 🧬 **11. Modelos Variantes de Turing**

Mesmo com alterações, o poder de computação mantém-se surpreendentemente igual.

### 🧯 11.1 Porquê estudar variantes?

As variantes da Máquina de Turing servem para:

* Simplificar certos algoritmos
* Tornar construções mais eficientes
* Modelar outras formas de computação

Mas **nenhuma dessas variantes é mais poderosa** em termos de linguagens aceites — todas reconhecem as mesmas linguagens **recursivamente enumeráveis**.

### 💽 11.2 Máquina de Turing com Múltiplas Fitas

Em vez de uma única fita, temos várias **fitas independentes**, cada uma com a sua cabeça de leitura/escrita.

* A função de transição agora lê e escreve em **várias posições ao mesmo tempo**
* Ajuda a escrever **algoritmos mais rápidos e mais organizados**

💡 Apesar de parecer mais poderosa, **toda MT de múltiplas fitas pode ser simulada por uma MT de fita única**, apenas com algum esforço e reorganização dos dados na fita.

### 🌌 11.3 Máquina de Turing Não-Determinista

Semelhante a um NFA, a máquina pode:

* Ter **várias opções** de transição para a mesma combinação de estado e símbolo
* Explorar **múltiplos caminhos computacionais** em simultâneo (conceitualmente)

💡 Uma MT não-determinista **aceita uma string se algum dos ramos levar ao estado de aceitação**

**Teorema:** Toda MT não-determinista pode ser simulada por uma MT determinista  
➡️ Isto mostra que, em termos de **poder computacional**, são equivalentes.

### 🌐 11.4 Máquina de Turing Universal (MTU)

Uma MTU é uma máquina que consegue **simular qualquer outra Máquina de Turing**. Basta dar-lhe:

* A descrição codificada de outra MT
* A entrada para essa máquina

💡 É o conceito teórico por trás de **qualquer computador moderno**: um dispositivo capaz de executar instruções arbitrárias (programas).

📦 O input da MTU é algo como:

〈M〉 # w

Ou seja: a descrição de uma máquina + uma string de entrada.

### 🚫 11.5 Máquina com Fita Infinita em Ambas as Direções

* Permite que a cabeça se mova para **esquerda ilimitadamente**
* Facilita a programação de certas máquinas

💡 Mas mais uma vez, qualquer MT com fita bi-infinita pode ser **simulada por uma com fita normal**, com codificação adequada.

### 💡 11.6 Outras variantes curiosas

* **Autómatos Linearmente Limitados (LBA):**  
  Subconjunto das MTs que **só podem usar espaço proporcional ao tamanho da entrada**
  + Muito usados em **linguagens sensíveis ao contexto**
  + Correspondem à classe **PSPACE**
* **Máquinas de Post:**  
  Sistema de regras com uma lista de instruções e manipulação de strings — **equivalente a MTs**, mas mais próxima da lógica simbólica

### 📌 Resumo Visual

| Variante | Diferença | Poder Computacional |
| --- | --- | --- |
| Múltiplas fitas | + organização | Igual |
| Não-determinista | múltiplos caminhos | Igual |
| Universal | simula outras máquinas | Igual |
| Bi-infinita | fita sem limites | Igual |
| LBA | espaço limitado | Menor |
| Post | regras simbólicas | Igual |

### 🧪 Experimenta tu mesmo/a!

1. Desenha um esquema de uma MT com **duas fitas** para inverter uma string.
2. Pensa: como podes codificar uma máquina qualquer (o seu "ADN") para que seja lida por uma **máquina universal**?

### 🧠 Conclusão

As variantes da Máquina de Turing são ferramentas valiosas para:

* **Modelar** computações mais naturalmente
* **Explorar limites teóricos**
* **Desenvolver algoritmos eficientes**

Mas o mais surpreendente é isto: **nenhuma dessas variantes tem mais poder do que o modelo base.** A Máquina de Turing simples já é suficiente para representar tudo o que é computacionalmente possível!

## 🧠 Parte IV – Turing, Computabilidade e Modelos Avançados

# ⚖️ **12. Decidibilidade e Indecidibilidade**

Até onde podem ir os algoritmos? E onde está o muro intransponível da computação?

### ✅ 12.1 Problemas Decidíveis

Um problema é **decidível** (ou **recursivo**) se existe **uma Máquina de Turing que o resolve para todas as entradas** — ou seja, que **termina sempre** com uma resposta “sim” ou “não”.

🔍 Exemplo clássico:

Este problema é decidível porque:

* Podemos **simular o DFA M** na string w
* O processo **termina sempre**, dado que o DFA é finito

Outros problemas decidíveis:

* Verificar se uma linguagem reconhecida por DFA é vazia
* Verificar se dois DFAs são equivalentes
* Verificar se uma CFG gera uma string w

### 🚫 12.2 Problemas Indecidíveis

Um problema é **indecidível** se **não existe Máquina de Turing alguma que o resolva para todas as entradas**.  
❗ Ou seja: **não há algoritmo que funcione para todos os casos**.

💣 O exemplo mais famoso:

Este é o **problema da aceitação**: “Dada uma Máquina de Turing e uma string , será que aceita ?”

🔴 Não há forma geral de saber — e isso **foi provado matematicamente** com um argumento genial: a **diagonalização**.

### 💡 12.3 A Prova por Diagonalização

Inspirada nos teoremas da incompletude de Gödel, a diagonalização é uma técnica para provar **que certas linguagens não podem ser reconhecidas**.

A ideia central:

* Enumeramos todas as máquinas de Turing possíveis (como programas)
* Criamos uma linguagem que **faz o oposto** daquilo que essas máquinas fariam
* Mostramos que **essa linguagem não está na enumeração**

➡️ Conclusão: **há mais linguagens do que máquinas** → algumas são indecidíveis.

### 🤯 12.4 Outros Exemplos de Problemas Indecidíveis

| Problema | Descrição |
| --- | --- |
|  | Saber se uma MT pára em certa entrada |
| Equivalência de MTs | Saber se duas MTs reconhecem a mesma linguagem |
| Vacuidade de MT | Saber se a linguagem de uma MT é vazia |
| Validade de programas (verificação completa) | Saber se um programa cumpre uma especificação |

💡 Todos estes problemas **não têm solução geral algorítmica** — mesmo com fita infinita, tempo ilimitado e poder de Turing!

### 📜 12.5 A Correspondência de Post

A **Correspondência de Post (PCP)** é um problema clássico que também é indecidível.  
É muitas vezes usado como ponto de partida para provar a indecidibilidade de outros problemas.

#### Enunciado:

Dado um conjunto de pares de strings, será que existe uma sequência de índices tal que as duas concatenações resultantes sejam iguais?

💣 Parece simples, mas não é! Não existe algoritmo que resolva todos os casos.

### 📌 Resumo Visual

| Conceito | Explicação |
| --- | --- |
| Decidível | A MT termina sempre e dá resposta correta |
| Indecidível | A MT não consegue decidir para todos os casos |
|  | Problema da aceitação de uma MT (indecidível) |
| Diagonalização | Técnica para provar indecidibilidade |
| PCP | Problema clássico indecidível baseado em pares |

### 🧪 Experimenta tu mesmo/a!

1. Tenta explicar por que **não há** uma função validaPrograma(p) que diga se o programa p termina ou não.
2. Imagina que consegues resolver . Que **paradoxo** isso criaria?

## 🧠 Parte IV – Turing, Computabilidade e Modelos Avançados

# 🧩 **13. Funções Recursivas e o Lado Matemático da Computação**

Uma viagem ao coração da lógica computável… onde os algoritmos ganham forma matemática.

### 🧠 13.1 O que é uma função computável?

Uma função é **computável** se existe uma forma mecânica (algoritmo ou máquina de Turing) de a calcular.

➡️ O estudo das **funções recursivas** surgiu como tentativa de capturar *todas as funções que podem ser efetivamente computadas*, e revelou-se equivalente ao modelo da Máquina de Turing!

### ⚙️ 13.2 Funções Primitivas Recursivas

Estas funções são construídas a partir de:

* Funções básicas:
  + (sucessor)
  + Projeções:

E combinadas com:

* **Composição** de funções
* **Recursão primitiva**

💡 Exemplos:

* Soma: definida com base no sucessor
* Multiplicação: usando soma recursiva

Estas funções **são sempre computáveis** e **terminam sempre**.

### 🔁 13.3 Funções μ-Recursivas (ou Generalizadas)

Para capturar **funções que não estão cobertas pela recursão primitiva**, introduzimos o **operador μ (minimização)**:

* : retorna o menor valor de tal que a função dá 0.

⚠️ Agora, **a computação pode não terminar** — ou seja, estas funções abrangem também funções **parcialmente computáveis**.

💡 As **funções μ-recursivas** são **equivalentes ao poder da Máquina de Turing**!

### 🧨 13.4 A Função de Ackermann

Um clássico exemplo de função **não-primitiva-recursiva**, mas **computável**:

🧠 Cresce mais depressa do que qualquer função primitiva recursiva.  
📈 A sua complexidade mostra como **há funções simples de definir, mas extremamente difíceis de calcular**.

### 📏 13.5 Gödelização e Números de Gödel

Para transformar **símbolos e fórmulas** em **números inteiros**, usa-se a técnica de Gödelização — codificação matemática que permite representar lógica simbólica dentro da aritmética.

👉 Fundamental na prova dos **teoremas da incompletude**.

### 🧮 13.6 O Teorema de Rice

“**Qualquer propriedade não-trivial sobre a linguagem reconhecida por uma Máquina de Turing é indecidível.**”

💥 Isso significa que **não podemos criar um programa que diga, em geral, se outro programa aceita só strings palíndromas, ou se reconhece uma linguagem finita, etc.**

É uma **bomba teórica**, pois mostra os **limites absolutos da análise de software**.

### 📐 13.7 A Diagonalização de Cantor

Método utilizado para provar que **existem mais linguagens do que máquinas**.  
Foi originalmente usado para provar que os **reais são incontáveis**, mas serve também para mostrar:

* Nem todas as linguagens são recursivamente enumeráveis
* Há problemas que **nenhuma máquina pode sequer enumerar**

### 🧠 13.8 λ-Cálculo e Computação

O **λ-cálculo** é outro modelo formal de computação, baseado exclusivamente em:

* Abstração de funções
* Aplicação de funções a argumentos

🧠 É base teórica de muitas linguagens de programação (como Haskell e partes do JavaScript).

💡 Apesar de muito diferente da Máquina de Turing, **é equivalente em poder computacional**!

### 📌 Resumo Visual

| Conceito | Significado |
| --- | --- |
| Primitiva recursiva | Função total e sempre terminável |
| μ-recursiva | Pode não terminar; inclui todas as computáveis |
| Ackermann | Exemplo de função não-primitiva |
| Teorema de Rice | Qualquer propriedade da linguagem de uma MT é indecidível |
| Diagonalização | Prova da existência de problemas não enumeráveis |
| Gödelização | Codifica símbolos como números |
| λ-cálculo | Modelo computacional baseado em funções |

### 🧪 Experimenta tu mesmo/a!

1. Constrói uma função recursiva para calcular com composição e recursão.
2. Tenta simular a função de Ackermann para — e prepara-te para muitas chamadas recursivas!
3. Pergunta-te: será possível escrever um programa que diga se outro termina sempre? (dica: não 😅)

## 🧠 Parte IV – Turing, Computabilidade e Modelos Avançados

# 🧩 **13. Funções Recursivas e o Lado Matemático da Computação**

Como podemos formalizar o que é “computável”? E até onde vão os limites da computação, vistos pela lente da matemática?

### 📍 13.1 O Conceito de Função Computável

Uma função é **computável** se existe um **algoritmo bem definido** que, em tempo finito, devolve o resultado correto para todas as entradas válidas.  
Estas funções são o coração da computação — **tudo o que um programa pode fazer é, no fundo, calcular funções**.

🧠 Mas como podemos representar, com rigor matemático, uma função computável?

### 🧱 13.2 Funções Primitivas Recursivas

As **funções primitivas recursivas** são funções *totais* que:

* **Terminam sempre**
* São definidas a partir de um conjunto pequeno de funções básicas:
  + **Função zero**:
  + **Função sucessora**:
  + **Projeções**:

➡️ E combinadas com dois mecanismos:

* **Composição** (encadeamento de funções)
* **Recursão primitiva** (definição em termos de si própria, mas sempre com base em um caso base)

🧮 Exemplo:  
A multiplicação pode ser definida recursivamente com base na adição.

### ⚠️ 13.3 Limitações: Precisamos de mais...

As funções primitivas recursivas **não capturam todas as funções computáveis**.  
➡️ Precisamos de introduzir um novo operador: **minimização** ou operador μ.

### 🔄 13.4 Funções μ-Recursivas

O operador μ permite definir funções com **busca não limitada**:

🔍 O resultado é o menor valor de tal que .  
Mas... e se **nunca houver tal valor**? 🤯

➡️ Aqui nasce a computação **parcial** — ou seja, **funções que podem não terminar**.

🎯 Resultado importante:

**Funções μ-recursivas = Funções computáveis pelas Máquinas de Turing**

### 📈 13.5 A Função de Ackermann

Exemplo clássico de função **computável mas não primitivamente recursiva**.

💥 Cresce mais depressa que qualquer função primitiva recursiva.  
Usada em teoria da complexidade para demonstrar **explosões combinatórias** e **limites de avaliação recursiva**.

### 🔢 13.6 Números de Gödel (Gödelização)

Para representar símbolos e fórmulas matemáticas como números inteiros (para os usar em funções aritméticas), usamos **Gödelização**:

* Cada símbolo recebe um número primo único.
* As sequências são codificadas como produtos primos elevados a potências.

➡️ Permite representar linguagens formais, autómatos ou programas como **números naturais**.  
🧠 Essencial na prova dos **Teoremas da Incompletude**.

### 🧪 13.7 O Teorema de Rice

**“Todas as propriedades não-triviais sobre a linguagem reconhecida por uma Máquina de Turing são indecidíveis.”**

Ou seja:

* Queres saber se uma MT **reconhece apenas strings palíndromas**? → indecidível.
* Se a linguagem é **vazia** ou **finita**? → indecidível, salvo casos especiais.

Este teorema estabelece **um limite absoluto ao que conseguimos descobrir automaticamente sobre algoritmos**.

### 🔁 13.8 A Diagonalização de Cantor

Técnica usada para provar:

* Que **existem mais funções** do que algoritmos para as calcular.
* Que **nem todas as linguagens** são recursivamente enumeráveis.

🧠 Inspira-se no argumento de Cantor que mostra que o conjunto dos números reais é não-enumerável.

Aplicação prática:

Mostra que **há problemas que nenhuma máquina poderá sequer listar** todas as strings válidas da sua solução.

### 🧩 13.9 λ-Cálculo: O modelo funcional da computação

* Sistema formal baseado em **funções anónimas** (lambda)
* Usa **abstração**, **aplicação** e **redução**

Embora muito diferente da Máquina de Turing:

É **equivalente em poder computacional** — ou seja, tudo o que se pode fazer com uma MT, pode fazer-se com λ-cálculo… e vice-versa.

💡 Base teórica de linguagens como Haskell, Scheme ou mesmo partes do JavaScript moderno.

### 📌 Resumo Tablado

| Conceito | Explicação | Termina sempre? | Computável? |
| --- | --- | --- | --- |
| Primitiva recursiva | Base + composição + recursão | ✅ | ✅ |
| μ-recursiva | Inclui busca ilimitada (minimização) | ❌ | ✅ |
| Ackermann | Exemplo não-prim. recursiva | ✅ | ✅ |
| Gödelização | Codifica fórmulas como inteiros | - | - |
| Teorema de Rice | Nenhuma propriedade útil da linguagem é decidível | ❌ | - |
| Diagonalização | Prova que há problemas não computáveis | ❌ | ❌ |
| λ-cálculo | Modelo alternativo, funcional | ✅/❌ | ✅ |

### 🧪 Desafios para ti!

1. Tenta definir a função usando apenas funções primitivas.
2. Imagina que queres saber se um programa p aceita apenas palíndromos. Podes criar um verificador? (Spoiler: o Teorema de Rice diz que não 😅)
3. Explora o crescimento de da função de Ackermann. Vais ficar boquiaberto!

## 🔍 **O que é a Diagonalização?**

A **diagonalização** é uma **técnica de prova por contradição** que mostra que **não é possível listar completamente certos conjuntos infinitos**, como o conjunto dos números reais ou das linguagens sobre um alfabeto.

### 👨‍🏫 Analogia intuitiva: a lista infinita de combinações

Imagina que tentas **enumerar todas as sequências infinitas de 0s e 1s**:

s₁ = 0 0 1 1 0 0 1 1 ...  
s₂ = 1 0 0 1 0 1 0 1 ...  
s₃ = 1 1 1 1 1 1 1 1 ...  
s₄ = 0 1 0 1 0 1 0 1 ...  
⋮

Tenta agora construir uma **nova sequência** , escolhendo **um novo bit diferente da diagonal da tabela**:

* : diferente de s₁\_1
* : diferente de s₂\_2
* : diferente de s₃\_3
* ...

Esta nova sequência **difere de todas as listadas**, pelo menos numa posição — ou seja, **não está na lista!**  
💥 Conclusão: **o conjunto total não é enumerável.**

## 🧠 Aplicação à Teoria da Computação

Agora vamos aplicar esta ideia ao **conjunto das linguagens** versus o **conjunto das máquinas de Turing**.

### 🎲 Passo 1: Representar máquinas como strings

Sabemos que:

* Cada Máquina de Turing pode ser codificada como uma string (número de Gödel, código binário, etc.)
* Logo, o conjunto de todas as MTs é **enumerável**

Vamos então listar todas as máquinas possíveis:

M₁, M₂, M₃, M₄, ...

E todas as strings da linguagem (alfabeto binário, por exemplo):

w₁ = "", w₂ = "0", w₃ = "1", w₄ = "00", w₅ = "01", ...

Agora criamos uma **tabela de aceitação**:

|  | w₁ | w₂ | w₃ | w₄ | w₅ | ... |
| --- | --- | --- | --- | --- | --- | --- |
| M₁ | 1 | 0 | 1 | 1 | 0 | ... |
| M₂ | 0 | 1 | 1 | 0 | 0 | ... |
| M₃ | 1 | 1 | 1 | 1 | 1 | ... |
| ... |  |  |  |  |  |  |

Onde:

* 1 = máquina aceita a string
* 0 = máquina rejeita a string ou não termina

### 🧨 Passo 2: Construir uma linguagem diagonal

Agora construímos a linguagem , tal que:

Ou seja, para cada diagonal , incluímos na linguagem **se não a aceitar**.

👉 Esta linguagem **difere da linguagem de cada**  exatamente em .

💥 Conclusão: **nenhuma máquina da lista reconhece**   
Mas acabámos de construir … então **existe uma linguagem que não é reconhecida por nenhuma MT** na enumeração.

➡️ Prova-se assim que **existem linguagens que não são recursivamente enumeráveis**!

## 📚 Interpretação Profunda

O argumento da diagonalização mostra que:

* **O conjunto das máquinas de Turing é enumerável** (contável)
* **O conjunto das linguagens possíveis é não-enumerável** (incontável)
* Logo: **existem mais linguagens do que máquinas** → há linguagens que **nenhuma máquina pode reconhecer**

Este resultado estabelece **um limite absoluto da computação**: **há problemas que não são apenas indecidíveis — são inalcançáveis.**

## 📌 Resumo Visual

| Conceito | Explicação |
| --- | --- |
| Diagonalização (Cantor) | Técnica que mostra não-enumerabilidade |
| Aplicação em computação | Prova que nem todas as linguagens são reconhecíveis |
|  | Linguagem diagonal construída artificialmente |
| Conclusão | Existe linguagem que nenhuma MT reconhece |

### 🧪 Para experimentar:

1. Constrói a tua própria tabela com 4 máquinas e 4 strings simples.
2. Marca quem aceita o quê.
3. Constrói a linguagem diagonal: escolhe o contrário da aceitação em cada célula da diagonal!

## 🚀 Parte V – Complexidade Computacional

# 🧩 **14. Complexidade de Tempo e Espaço**

Afinal, o que significa dizer que um problema é "difícil" para um computador?

Quando um algoritmo **resolve um problema**, o passo seguinte é perguntar:

* *Quão rapidamente?*
* *Com quanto espaço de memória?*

Estes dois recursos — **tempo** e **espaço** — são a base da teoria da **complexidade computacional**.

### ⏱️ 14.1 Noções Fundamentais

#### 📐 Medidas de Complexidade

* **Tempo**: número de passos (ou transições) que a Máquina de Turing faz.
* **Espaço**: número de células da fita usadas.

Medimos a complexidade **em função do tamanho da entrada** .  
Exemplo:  
Um algoritmo que verifica se um número tem fatores é mais lento para entradas maiores.

### 🔢 14.2 Notações Assintóticas

Estas notações ajudam a **classificar algoritmos de forma independente da máquina específica**.

* **Big-O ()**: Limite superior. Ex: tempo máximo de execução.
* **Ω (ômega)**: Limite inferior. Garante que não pode ser mais rápido.
* **Θ (teta)**: Tempo “exato”, quando inferior e superior coincidem.
* **o (pequeno-o)**: Cresce *estritamente menos* que
* **ω (pequeno-ômega)**: Cresce *estritamente mais* que

🔎 Exemplo prático:

Se um algoritmo tem tempo O(n²), significa que, no pior caso, o tempo cresce no máximo como n².

### 🧮 14.3 Classes de Complexidade

As classes agrupam problemas segundo os recursos que exigem para serem resolvidos.

#### 💨 **Classe P (Polynomial Time)**

* Problemas resolvidos em tempo polinomial.
* Ex: ordenação, procura binária, multiplicação de matrizes...

Computacionalmente *tratáveis*. Podemos implementá-los e confiar neles para grandes entradas.

#### 💎 **Classe NP (Non-deterministic Polynomial Time)**

* Soluções podem ser **verificadas** em tempo polinomial.
* Ex: Sudoku, Mochila, SAT (satisfiabilidade booleana)

⚖️ Aqui surge a **grande pergunta da computação moderna**:

Será que **P = NP**?

Ou seja: se conseguimos verificar soluções rapidamente, conseguimos também **encontrá-las rapidamente**?

💰 Esta é a **questão do milhão de dólares** do Clay Mathematics Institute.

### 🧠 14.4 Problemas NP-Completos

São os **problemas mais difíceis de NP**.  
Se resolves **um deles em tempo polinomial**, resolves **todos os outros**.

✅ Exemplo clássico:  
**SAT** – existe uma atribuição de verdade que torna uma fórmula booleana verdadeira?

🔁 Outros problemas **NP-completos**:

* Problema da mochila (Knapsack)
* Caminho Hamiltoniano
* Cobertura de vértices

🧠 A maioria deles está relacionada com **combinatória**, **lógica** e **otimização**.

### 🔄 14.5 Reduções Polinomiais

Transformar um problema em outro sem aumentar muito o tempo de execução.

Se o problema A se reduz ao problema B, e **B é fácil**, então **A também é**.

* Escreve-se:
* Essencial para provar que um problema é **NP-completo**

🧩 Ferramenta base para **análise de complexidade entre problemas**

### 📘 14.6 Teorema de Cook–Levin

📌 Mostra que:

SAT é **NP-completo** — ou seja, é tão difícil quanto qualquer outro problema em NP.

Foi o **primeiro problema NP-completo descoberto** e abriu caminho para a teoria moderna da complexidade.

📜 Prova-se através de uma transformação de qualquer Máquina de Turing para uma fórmula booleana.

### 💾 14.7 Complexidade de Espaço

Nem só de tempo vive um algoritmo!

* **PSPACE**: Problemas resolvidos com **espaço polinomial**
* **L (Logspace)**: Usa apenas **espaço logarítmico**
* **NL (Nondeterministic Logspace)**: Versão não-determinística de L

⚠️ Algumas relações:

L ⊆ NL ⊆ P ⊆ NP ⊆ PSPACE

Mas… **algumas inclusões ainda não se sabem se são estritas**!  
Exemplo: Será que **P = PSPACE**? Mistério!

### 📌 Tabela Resumo

| Classe | Definição | Exemplos | Abertas? |
| --- | --- | --- | --- |
| P | Tempo polinomial determinístico | Ordenar, somar | - |
| NP | Verificável em tempo polinomial | Sudoku, SAT | ❓ P = NP? |
| NP-Completo | O mais difícil de NP | SAT, 3-CNF-SAT | ❓ |
| PSPACE | Espaço polinomial | Jogos, Lógica Quantificada | ❓ |
| L | Espaço logarítmico | Verificação simples | - |
| NL | Espaço logaritmo não-determinístico | Caminho em grafos | ❓ L = NL? |

### 🧪 Desafios para ti:

1. Pensa num problema real (como resolver um puzzle de Sudoku). A verificação é rápida? E a resolução?
2. Consegues imaginar como transformar um problema de decisão em outro (ex: SAT → 3-SAT)?
3. Porque achas que P = NP é uma questão tão importante para criptografia, IA e otimização?

## 🔧 Parte VI – Aplicações e Temas Avançados

# 🧵 **15. Gramáticas e Compiladores**

Como é que o teu código fonte é lido, compreendido e transformado por uma máquina?

### 📍 15.1 O que é um compilador?

Um **compilador** é um programa que **traduz código fonte** (escrito por humanos) para **código máquina** (executável por computadores).  
Este processo envolve várias fases, das quais destacamos:

1. **Análise léxica**
2. **Análise sintática**
3. **Análise semântica**
4. **Geração de código intermediário**
5. **Otimização**
6. **Geração de código final**

Vamos focar-nos nas **duas primeiras**, que são diretamente ligadas à **teoria das linguagens formais**.

### 🔠 15.2 Análise Léxica (Scanner)

Responsável por **ler a sequência de caracteres do código fonte** e transformá-los em **tokens** — as unidades básicas da linguagem.

🔤 Exemplo:  
Para o código:

while (x < 10) { x = x + 1; }

A análise léxica transforma isto em:

[WHILE][LPAREN][ID:x][LT][NUM:10][RPAREN][LBRACE]...

✅ Utiliza:

* **Expressões regulares**
* **Autómatos finitos deterministas (DFA)**

🛠️ Ferramentas como **Lex** e **Flex** implementam esta fase com base direta nas linguagens regulares.

### 🌳 15.3 Análise Sintática (Parser)

A análise sintática verifica se a sequência de tokens segue as **regras gramaticais da linguagem**.

➡️ Usa **gramáticas livre de contexto (CFG)**  
➡️ Produz **árvores de derivação (parse trees)**

Exemplo de regras:

stmt → while ( cond ) stmt  
cond → expr < expr  
expr → expr + expr | ID | NUM

🧠 Se a sequência de tokens **não puder ser gerada pela gramática**, o parser devolve erro de sintaxe.

### 🔄 15.4 Parsing Top-Down vs Bottom-Up

#### 🔼 Top-Down

* Tenta derivar a entrada a partir do símbolo inicial.
* Ex: **LL(1) parsing**
* Simples e previsível, mas **limitado** (não lida bem com ambiguidade ou recursividade à esquerda)

#### 🔽 Bottom-Up

* Reconstrói a árvore **a partir das folhas**.
* Ex: **LR parsing**
* Muito mais poderoso e **usado em compiladores reais**

### ⚙️ 15.5 Algoritmos Famosos

#### ✅ LL(1)

* Usa **tabelas preditivas**
* Funciona bem com **gramáticas não ambíguas e fatoradas**
* Fácil de implementar

#### ✅ LR(0), SLR, LALR

* Usam **autómatos com pilha**
* Analisam com base no **estado atual + símbolo de entrada**
* **LALR(1)** é amplamente usado em compiladores modernos (ex: yacc, bison)

### 🛠️ 15.6 Ferramentas do mundo real

| Ferramenta | Fase | Base teórica |
| --- | --- | --- |
| **Lex/Flex** | Análise léxica | ERs e DFAs |
| **Yacc/Bison** | Análise sintática | CFGs e LR parsing |
| **ANTLR** | Parser generator avançado | LL(\*) parsing |

### 🎓 15.7 Do formalismo à prática

| Formalismo teórico | Aplicação em compiladores |
| --- | --- |
| Expressões regulares | Tokens e scanners |
| DFA/NFA | Reconhecimento léxico |
| CFG | Sintaxe da linguagem |
| PDA | Parsing bottom-up |
| Árvores de derivação | Construção de ASTs |
| Ambiguidade | Prevenção de erros de compilação |

💡 Compiladores são, na prática, **implementações cuidadosas e otimizadas** da Teoria da Computação.

### 📌 Resumo Visual

| Conceito | Ligação com a Teoria |
| --- | --- |
| Token | Linguagem regular |
| Scanner | Autómato finito |
| Parser | CFG |
| Análise sintática | PDA |
| Árvores de derivação | Derivações à esquerda/direita |
| Parsing LL/LR | Algoritmos com pilha |

### 🧪 Desafios para ti:

1. Escreve uma expressão regular que reconheça **identificadores** como nome123.
2. Define uma CFG para expressões aritméticas simples com + e \*.
3. Constrói uma **árvore de derivação** para while (x < 10) x = x + 1;

## 🔧 Lex/Flex e Yacc/Bison – Onde a Teoria Ganha Vida!

### 🧩 1. Lex e Flex – O Scanner Gerador

**Lex** (ou a versão moderna **Flex**) é uma ferramenta que automatiza a **análise léxica**, ou seja, transforma um fluxo de caracteres em **tokens** com significado.

#### 🔍 Como funciona?

Tu defines regras do tipo:

[0-9]+ { return NUM; }  
"if" { return IF; }  
[a-zA-Z]+ { return ID; }

💡 Cada linha descreve um **padrão (expressão regular)** e o **código associado** que é executado quando esse padrão é reconhecido.

#### 🔁 Por detrás do cortinado:

* Lex transforma essas REs num **AFN**.
* Converte esse AFN num **AFD**.
* Gera código C (ou C++) que implementa esse AFD.

Ou seja: **autómatos finitos a funcionar na prática!** 🎯

### 🧠 2. Yacc e Bison – O Mestre da Análise Sintática

**Yacc** (*Yet Another Compiler Compiler*) e **Bison** (versão moderna GNU) geram parsers **bottom-up**, normalmente do tipo **LALR(1)**.

#### 📐 A estrutura típica:

%token NUM PLUS TIMES  
  
%%  
expr : expr PLUS term  
 | term ;  
  
term : term TIMES factor  
 | factor ;  
  
factor : NUM ;  
%%

Cada regra é como uma **produção gramatical** de uma **gramática livre de contexto**.

🔧 A cada produção associamos ações em C/C++ para construir árvores, avaliar expressões, etc.

### 🔗 Como se ligam Lex e Yacc?

1. O **Lex** analisa o texto de entrada e devolve **tokens** para o **Yacc**.
2. O **Yacc** usa esses tokens para aplicar regras gramaticais e **validar a estrutura da linguagem**.
3. Ambos partilham um ficheiro de cabeçalho com os tokens (tokens.h).

#### Exemplo de integração:

lex scanner.l # gera lex.yy.c  
yacc -d parser.y # gera y.tab.c e y.tab.h  
gcc lex.yy.c y.tab.c -o compilador

### 🧪 Um mini-exemplo funcional

**scanner.l** (Lex):

%{  
#include "y.tab.h"  
%}  
%%  
[0-9]+ { yylval = atoi(yytext); return NUM; }  
"+" { return PLUS; }  
. { return yytext[0]; }  
%%

**parser.y** (Yacc):

%token NUM PLUS  
%%  
expr : expr PLUS NUM { printf("Soma: %d\n", $1 + $3); }  
 | NUM { $$ = $1; }  
;  
%%

📦 O que este par faz:

* Lê somas como 3+5+2
* Avalia-as na hora, construindo implicitamente uma **árvore de derivação** com ações semânticas.

### 🔬 Relação com a Teoria da Computação

| Conceito teórico | Implementação prática |
| --- | --- |
| Expressões regulares | Lex / Flex |
| Autómatos finitos | DFA interno do Lex |
| Gramáticas LFC | Yacc / Bison |
| Autómatos com pilha | Parser LR (PDA implícito) |
| Árvores de derivação | Ações nas regras do Yacc |
| Ambiguidade gramatical | Erros em parsing LR/LL |

### 🤓 Curiosidade histórica

* **Lex** foi criado nos anos 70 por *Eric Schmidt* (sim, o futuro CEO da Google).
* **Yacc** foi criado por *Stephen Johnson* como ferramenta para o UNIX.

Hoje, **Flex** e **Bison** são usados por compiladores reais como o **GCC**!

### 📚 Sugestão para o teu tutorial

Podes incluir:

* Uma secção prática com **exemplo completo** (scanner + parser)
* Laboratório: construir um interpretador para expressões aritméticas
* Desafio: adicionar \* e () à linguagem

## 🔧 Parte VI – Aplicações e Temas Avançados

# 🌌 **16. Temas Avançados e Exploratórios**

### 🧪 16.1 Autómatos Probabilísticos (PFA)

E se os autómatos tomassem decisões ao acaso?

Um **autómato probabilístico** (PFA) é como um autómato finito... mas com **transições associadas a probabilidades**. Em vez de escolher uma única transição de forma determinista ou não-determinista, o PFA escolhe **aleatoriamente**, com base numa distribuição de probabilidades.

#### 🔁 Definição intuitiva:

* Um conjunto de estados, alfabeto, função de transição probabilística.
* Existe uma **probabilidade associada** a cada transição.
* A palavra é **aceite** se a soma das probabilidades dos caminhos que levam a um estado final **excede um limiar**.

#### 🔍 Aplicações:

* Modelos de **linguagem natural** (ex: reconhecimento de voz)
* **Verificação probabilística** de sistemas
* **Redes de Markov discretas**

⚠️ Observação: Nem todas as linguagens regulares são reconhecidas por PFAs com erro arbitrariamente pequeno!

### 🧬 16.2 Autómatos Celulares

Computação distribuída em grade – o poder das regras locais a gerar padrões globais.

Um **autómato celular** é um modelo composto por uma **malha de células**, cada uma com um **estado** que evolui ao longo do tempo consoante **regras locais de vizinhança**.

#### 🎨 Exemplo famoso:

**Regra 30** e **Jogo da Vida (Game of Life)** de Conway.

#### 🧠 Aplicações:

* Simulação de **sistemas biológicos**
* Estudo de **comportamentos emergentes**
* **Criptografia e pseudo-aleatoriedade**
* **Física computacional** (ex: simulação de fluidos)

#### 💡 Ideia central:

Padrões complexos podem emergir de **regras simples** – uma verdadeira metáfora para a vida e a computação natural.

### 🧮 16.3 Gramáticas Matriciais

Quando a sequência das produções também importa.

As **gramáticas matriciais** estendem as gramáticas clássicas ao permitirem **sequências específicas de produções agrupadas**.

#### 🧱 Estrutura:

* Em vez de aplicar qualquer produção, é aplicada uma **matriz de produções** (um vetor ordenado).
* Linguagem gerada: qualquer cadeia que possa ser gerada por **uma sequência de matrizes aplicadas na ordem**.

#### ⚙️ Consequências:

* Linguagens mais poderosas que as livre de contexto.
* Relacionadas com **gramáticas contextuais** e **linguagens recursivamente enumeráveis**.

### 🌿 16.4 L-Systems (Sistemas de Lindenmayer)

Crescimento de plantas... gerado por gramáticas!

Desenvolvido para modelar **crescimento biológico**, um **L-system** é uma **gramática paralela**, onde **todas as produções aplicam-se em simultâneo** numa etapa.

#### 🌱 Exemplo simples:

A → AB   
B → A

Começando em A, obténs:

* A
* AB
* ABA
* ABAAB
* ...

#### 🧠 Aplicações:

* **Modelação botânica e fractal**
* **Gráficos por computador**
* **Simulação de crescimento natural**
* **Arte algorítmica**

### 📊 Comparação geral dos modelos

| Modelo | Características | Potência Computacional |
| --- | --- | --- |
| PFA | Transições com probabilidades | Entre DFA e Turing Machines |
| Autómato Celular | Regras locais paralelas | Pode simular Turing Machines |
| Gramática Matricial | Sequência ordenada de produções | Entre CFL e linguagens recursivas |
| L-System | Produções paralelas (simultâneas) | CFL ou mais (dependente do modelo) |

### 🧪 Desafios para ti

1. Constrói um PFA simples que aceite números binários divisíveis por 3 com 80% de certeza.
2. Simula um **autómato celular** que desenha um padrão em forma de árvore.
3. Define um L-System para modelar uma planta com galhos simétricos.
4. Escreve uma **gramática matricial** que gere a linguagem { aⁿbⁿcⁿ | n ≥ 1 }.

### 📌 Resumo da secção

Nesta parte, abrimos horizontes para **modelos não convencionais**, que:

* **Expandem os limites** da computação clássica
* Estão fortemente ligados à **natureza, probabilidade e paralelismo**
* São **inspiradores para investigação e criação artística**