Compiladores – Aula 1

1. Introdução

Um **compilador** é um tradutor que transforma um programa escrito em uma **linguagem de alto nível** (C, Java, Python) em uma **forma de mais baixo nível** (assembly, bytecode ou código de máquina), preservando o significado do programa.

Motivações para compilar

- **Desempenho**: execução nativa tende a ser mais rápida que interpretação.
- **Verificação antecipada**: erros de sintaxe e semântica são detectados antes da execução.
- **Portabilidade**: o mesmo código pode ser recompilado para várias arquiteturas.
- **Otimização**: melhor uso de CPU, memória e recursos do sistema.
- **Interoperabilidade**: código pode ser integrado a bibliotecas de diferentes linguagens.

Camadas clássicas

- **Front-end**: análise léxica, sintática e semântica; gera uma **representação intermediária (IR)**.
- **Middle-end**: aplica otimizações independentes de arquitetura (ex.: eliminação de código morto, propagação de constantes).
- **Back-end**: converte IR em instruções específicas da arquitetura alvo, gerando **código objeto**.

Exemplo prático (gcc)

```
gcc -S teste.c   # gera assembly
gcc -c teste.c   # gera objeto (.o)
gcc teste.c -o teste   # gera executável
```

Ferramentas do toolchain

- **Assembler (as)**: assembly → objeto (.o).
- **Linker (ld)**: combina objetos e bibliotecas → executável.
- **Loader (SO)**: carrega o executável na memória e inicia sua execução.

Exemplo ponta-a-ponta

```
int main() { return 2 + 3; }
```

Fluxo:

- Fonte em C → tokens (`int`, `main`, ...)
- AST → nó de retorno com soma
- IR (três endereços): `t1 = 2 + 3 ; return t1`
- Otimização: `return 5`
- Assembly: `mov eax,5 ; ret`

Recursos para estudo

- *Dragon Book*, Cap. 1 – Introduction
- GCC Online Documentation
- Tutorialspoint – Compiler Design Overview
- Neso Academy – Introduction to Compiler Design (YouTube)

---

2. Anatomia de um Compilador Convencional

Um compilador convencional é dividido em **módulos/fases**, cada um com responsabilidades bem definidas, que transformam progressivamente o código-fonte até chegar ao executável final.

Fluxo típico de compilação

```
Fonte → Lexer → Parser → Análise Semântica → IR → Otimização → Geração de Código → Objeto → Linkedição → Executável
```

Exemplo ponta-a-ponta

```c
int x = 5 + 3;
```

- Léxico: Tokens `INT ID = NUM + NUM ;`
- Sintaxe: AST `Assign(x, Add(5,3))`
- Semântica: tipos de `x`, `5` e `3` compatíveis
- IR (três endereços): `t1 = 5 + 3 ; x = t1`
- Otimização: `x = 8`
- Assembly: `mov r0,#8 ; str r0,[x]`
- Objeto/Link: símbolo `x` resolvido, relocação feita

Fases e saídas resumidas

- **Análise Léxica** → tokens
- **Análise Sintática** → AST
- **Análise Semântica** → AST anotada + tabela de símbolos
- **Representação Intermediária (IR)** → três endereços, SSA, LLVM IR
- **Otimização** → código melhorado
- **Geração de Código** → objeto
- **Linkagem** → executável

Tabela de símbolos

- Estrutura acessada por léxico, sintático e semântico.
- Mantém informações: identificadores, tipos, escopos, endereços.

Erros e diagnósticos

- Léxico: caractere inválido.

- Sintaxe: tokens fora de ordem, parêntese não fechado.
- Semântico: variável não declarada, incompatibilidade de tipos.

Relação com Hierarquia de Chomsky

- Lexer → linguagens regulares (Tipo 3).
- Parser → gramáticas livres de contexto (Tipo 2).
- Semântico → próximo de linguagens sensíveis ao contexto (Tipo 1).
- Geração de código → computação Turing-completa (Tipo 0).

Recursos para estudo

- *Dragon Book*, Cap. 2 – Structure of a Compiler
- Compiler Design Overview – Tutorialspoint
- Neso Academy – Compiler Structure (YouTube)

---

3. Analisador Léxico (Lexer)

O **lexer** (ou scanner) é a **primeira fase do compilador**, responsável por transformar o fluxo de **caracteres** em **tokens**. Ele usa **expressões regulares** e **autômatos finitos determinísticos (DFAs)** para reconhecer padrões.

Conceitos-chave

- **Token**: unidade léxica com significado (ex.: `(ID, "soma")`, `(NUM, 123)`).
- **Lexema**: trecho concreto do texto que corresponde ao token.
- **Padrão**: regra que define como identificar tokens (usualmente uma regex).

Exemplo prático em C (lexer manual simplificado):

```c
#define ID 1024
int isID(FILE *tape) {
  int head = getc(tape);
  if (isalpha(head)) {
    while (isalnum(head = getc(tape)));
    ungetc(head,tape);
    return ID;
  }
  ungetc(head,tape);
  return 0;
}
```

Esse código identifica identificadores (variáveis, nomes de funções) e retorna um token específico.

Lexer como coquetel de autômatos

- Implementado como a composição de diversos **DFAs**, cada um reconhecendo classes distintas de tokens (identificadores, números, operadores, palavras-chave, etc.).
- A execução escolhe o autômato válido com base no **maior casamento (maximal munch)**.

Exemplo em Flex (gerador de lexer)

```
%%
[
]+                /* ignora espaços */
"if"                   return IF;
[a-zA-Z_][a-zA-Z0-9_]* { yylval.str = strdup(yytext); return ID; }
[0-9]+                 { yylval.num = atoi(yytext); return NUM; }
.                      return *yytext; /* operadores */
%%
```

O Flex converte essas regras em um DFA eficiente.

Erros comuns detectados pelo lexer

- Caractere inválido (ex.: `@` em C).
- Literais malformados (ex.: `123abc`).
- Recuperação: consumir caractere problemático e seguir adiante, sempre registrando linha e coluna.

ASCII vs Unicode

- Em C clássico: cada `char` é apenas um número ASCII.
- Em linguagens modernas: lexers precisam lidar com Unicode, emojis e símbolos internacionais.

Recursos para estudo

- *Dragon Book*, Cap. 3 – Lexical Analysis
- Flex Manual
- Tutorialspoint – Lexical Analysis
- Neso Academy – Lexical Analysis (YouTube)

---

4. Analisador Sintático (Parser)

O **parser** é a segunda fase do compilador. Ele recebe os tokens do lexer e verifica se a sequência corresponde a uma **gramática livre de contexto (GLC)** da linguagem. Sua saída é geralmente uma **Árvore Sintática (Parse Tree)** e, em seguida, uma **Árvore Sintática Abstrata (AST)**, que organiza a estrutura do programa de forma hierárquica e simplificada.

Funções principais

- Detectar e reportar **erros sintáticos** (parênteses ausentes, tokens em ordem incorreta, ponto e vírgula faltando).
- Construir estruturas que representem a hierarquia do programa.
- Fornecer a base para a análise semântica.

Tipos de parser

- **Top-down** (ex.: LL(1), descida recursiva): deriva a partir do símbolo inicial, prevendo tokens à frente.
- **Bottom-up** (ex.: LR, LALR, SLR): parte da entrada e reduz até o símbolo inicial; usado em compiladores reais (GCC usa Bison/Yacc).

Exemplo de gramática e análise

```
stmt → ID = expr ;
expr → expr + term | term
term → ID | NUM | ( expr )
```

Entrada: `x = a + b;` Tokens: `ID(x) = ID(a) + ID(b) ;` Parse Tree mostra a atribuição, com expr → expr + term.

AST simplificada

```
Assign(
  ID(x),
  Add(ID(a), ID(b))
)
```

Erros sintáticos comuns

- Token inesperado (ex.: `if (x 5)` → falta operador).
- Estrutura incompleta (ex.: `while (x < 5` sem fechar parêntese).
- Ambiguidade: gramáticas mal definidas podem gerar múltiplas árvores possíveis.

Ferramentas práticas

- `yacc` e `bison`: geradores de parsers LR.
- `ANTLR`: gera parsers LL recursivos para várias linguagens.
- Visualizadores: jsmachines.github.io (simulador interativo de parsers).

Relação com teoria

- Baseia-se em **Gramáticas Livres de Contexto** (Tipo 2 na hierarquia de Chomsky).
- Conecta tokens regulares (Tipo 3) com estruturas semânticas mais complexas.

Recursos para estudo

- *Dragon Book*, Cap. 4 – Syntax Analysis
- Parsing – Wikipedia
- Neso Academy – Parsing Techniques (YouTube)

---

5. Analisador Semântico

O **analisador semântico** verifica se o programa não apenas segue a sintaxe correta, mas também se **faz sentido** de acordo com as regras da linguagem. Ele trabalha sobre a **AST** produzida pelo parser, consultando e atualizando a **tabela de símbolos**.

Principais responsabilidades

- Verificar **declaração e uso de identificadores** (variáveis, funções, tipos).
- Garantir **compatibilidade de tipos** em expressões e atribuições.
- Verificar **escopos**: símbolos só são válidos dentro de seus contextos.
- Checar **assinaturas de funções**: quantidade e tipo de argumentos.
- Confirmar **tipos de retorno** em funções.
- Inserir anotações de tipo na AST.

Exemplo prático em C

```
int a;
double b;
a = b + 3;
```

- Lexer: tokens (`int` , `a` , `;` , `double` , `b` , ...)
- Parser: AST para `a = b + 3`
- Semântico: detecta que `b + 3` é `double` ; atribuição para `a` (int) exige coerção.

Tabela de símbolos Estrutura fundamental que armazena informações sobre cada identificador:

- Nome
- Tipo
- Escopo
- Categoria (variável, função, constante)
- Atributos adicionais (posição de memória, parâmetros)

Erros típicos detectados

- Variável não declarada usada em expressão.
- Conversão de tipos inválida.
- Chamada de função com número ou tipos incorretos de argumentos.
- Retorno de tipo diferente do declarado.

Ligação com código da disciplina (lexer.c)

- O lexer detecta tokens.
- O parser organiza a estrutura sintática.
- O analisador semântico consulta a tabela de símbolos para validar coerência e significado.

Recursos para estudo

- *Dragon Book*, Cap. 5 – Semantic Analysis
- Tutorialspoint – Semantic Analysis
- Neso Academy – Semantic Analysis (YouTube)

6. Notação T

A **Notação T** é uma forma de representação intermediária baseada em **instruções de três endereços**. Ela é amplamente utilizada em materiais didáticos (como os slides da disciplina) por ser clara para visualizar a decomposição de expressões complexas em operações elementares.

Estrutura típica

```
Tn = op arg1, arg2
```

- `Tn`: variável temporária gerada automaticamente.
- `op`: operação aritmética ou lógica (+, -, *, /, <, etc.).
- `arg1`, `arg2`: operandos (podem ser variáveis, constantes ou temporários).

Exemplo (dos slides) Código-fonte em C:

```
a = b + c * d;
```

Notação T:

```
T1 = c * d
T2 = b + T1
a  = T2
```

Composição simples

- Cada operação gera uma nova variável temporária.
- Expressões grandes se tornam uma sequência linear de atribuições.
- Facilita o mapeamento para árvores de expressão e para código assembly.

Migração de arquitetura

- O mesmo conjunto de instruções em notação T pode ser traduzido para diferentes ISAs (Instruction Set Architectures).
- Exemplo: `T1 = c * d` gera instruções diferentes em x86, ARM e RISC-V, mas a IR permanece a mesma.

Bootstrapping e slides

- Nos slides, a notação T aparece também ligada ao conceito de **bootstrapping**: quando um compilador se compila usando sua própria linguagem, a notação T serve como camada intermediária.
- Exemplo clássico mostrado: etapas `C0` (assembly) → `C1` (na própria linguagem) → `C2` (otimizado), sempre gerando notação T no meio.

Otimizações possíveis

- *Constant folding*: reduzir expressões com valores fixos (`T1 = 2*3` → `T1=6`).
- *Common subexpression elimination*: reutilizar resultados já computados.

Comparação com representações modernas

- **TAC (Three Address Code)**: equivalente à notação T.
- **SSA (Static Single Assignment)**: cada variável tem uma única definição.
- **LLVM IR**: extensão moderna que se inspira em TAC/SSA.

Exemplo em LLVM IR (análogo ao slide de notação T)

```
%1 = mul i32 %c, %d
%2 = add i32 %b, %1
store i32 %2, i32* %a
```

Recursos para estudo

- *Dragon Book*, Cap. 6 – Intermediate Code Generation
- Three Address Code – GeeksforGeeks
- LLVM Language Reference
- Slides da Aula 1 – exemplos de notação T aplicados em expressões aritméticas e bootstrapping.

---

7. Código Realocável

Um **código realocável** (ou relocável) é um código objeto que pode ser carregado em **qualquer posição da memória** e ainda funcionar corretamente. Ele é fundamental para permitir que múltiplos programas coexistam na memória e que módulos compilados separadamente sejam unidos pelo linker.

Conceito

- Não contém endereços absolutos fixos.
- Usa referências simbólicas ou relativas que o linker/loader resolve.
- Garante portabilidade do módulo objeto dentro do espaço de endereçamento.

Exemplo em C

```
gcc -c exemplo.c -o exemplo.o   # gera relocável (objeto)
objdump -r exemplo.o            # mostra tabelas de realocação
readelf -r exemplo.o            # também exibe seções de realocação
```

Tipos de código

- **Absoluto**: endereços fixos, só funciona se carregado em posição específica.
- **Relocável**: ajustado dinamicamente pelo linker/loader.
- **Executável**: já está pronto para rodar, com endereços resolvidos.

Tabela de realocação

- Criada pelo assembler.
- Lista símbolos e posições no código onde endereços precisam ser ajustados.
- O linker usa essas informações para corrigir os endereços ao gerar o executável.

Exemplo prático de realocação Código fonte:

```
extern int y;
int x = 5;
int soma() { return x + y; }
```

- O objeto `.o` contém referência indefinida a `y`.
- A tabela de realocação marca esse ponto.
- O linker resolve quando encontrar `y` em outra unidade de tradução.

Importância

- Permite **modularidade** (vários arquivos .c → .o → binário).
- Suporte a **bibliotecas estáticas/dinâmicas**.
- Facilita **multitarefa** e **gerenciamento de memória**.

Recursos para estudo

- *Dragon Book*, Cap. 7 – Run-time Storage Organization
- `man objdump`, `man readelf`
- ELF Format – OSDev Wiki
- Relocation in Object Files – GeeksforGeeks

---

8. Controle de Versões em C

O **controle de versões** é indispensável em projetos de compiladores escritos em C (ou qualquer software complexo). Ele organiza a evolução do código, permite colaboração entre desenvolvedores e garante reprodutibilidade.

Por que usar?

- Histórico de alterações: voltar a estados anteriores do código.
- Trabalho em equipe: vários desenvolvedores contribuem sem sobrescrever o trabalho alheio.
- Criação de ramificações (branches): desenvolvimento de novas funcionalidades de forma isolada.
- Controle de versões estáveis (tags): marcar milestones do projeto.
- Integração com ferramentas de build (Makefile, CMake) e integração contínua (CI/CD).

Exemplo prático com Git

```
git init
git add lexer.c parser.c main.c
```

```
git commit -m "Implementa lexer inicial"
git checkout -b parser-dev   # cria branch específica para parser
git merge parser-dev          # integra branch ao projeto principal
git tag -a v1.0 -m "Versão inicial do compilador"
```

Makefile e integração Manter um Makefile versionado garante reprodutibilidade da compilação:

```
CC=gcc
CFLAGS=-Wall -g
OBJS=lexer.o parser.o main.o

compilador: $(OBJS)
    $(CC) $(CFLAGS) -o compilador $(OBJS)

%.o: %.c
    $(CC) $(CFLAGS) -c $<
```

Compilar com:

```
make
```

Ferramentas adicionais

- Hospedagem: GitHub, GitLab, Bitbucket.
- CI/CD: GitHub Actions, GitLab CI, Jenkins.
- Análise estática: `clang-tidy`, `cppcheck`.
- Padronização de código: `clang-format`.

Boas práticas

- Commits pequenos e descritivos.
- Usar branches para novas features.
- Revisão de código (pull requests).
- Documentar o processo de build no repositório.

Recursos para estudo

- Pro Git Book
- Makefile Tutorial
- GitHub Student Pack
- Atlassian Git Tutorials

---

9. Classificação na Hierarquia de Chomsky

A **Hierarquia de Chomsky** classifica linguagens formais em quatro níveis, cada um com poder expressivo diferente. Ela é fundamental para entender em qual classe se encaixa cada fase de um compilador.

Tipo	Descrição	Exemplo no compilador
0	Recursivamente Enumeráveis	Qualquer computação possível, compilador como um todo (Turing completo)
1	Sensíveis ao Contexto	Regras semânticas complexas (ex.: verificação de tipos, coerção, escopos)
2	Livres de Contexto (CFGs)	Analisador sintático (parser)
3	Regulares	Analisador léxico (lexer)

Exemplos práticos

- **Tipo 3 (Regulares)**: reconhecer identificadores (`[a-zA-Z_][a-zA-Z0-9_]*`), números (`[0-9]+`), operadores (`+`, `-`, `*`).
- **Tipo 2 (CFGs)**: validar expressões `if (expr) stmt else stmt`.
- **Tipo 1 (Sensíveis ao Contexto)**: garantir que variáveis são declaradas antes do uso e verificar compatibilidade de tipos.
- **Tipo 0 (Recursivamente Enumeráveis)**: o compilador como sistema completo, capaz de decidir ou simular qualquer computação.

Relação com fases do compilador

- Lexer → Tipo 3
- Parser → Tipo 2
- Analisador semântico → próximo de Tipo 1
- Gerador de código → Tipo 0

Recursos para estudo

- *Dragon Book*, Cap. 2 – Introdução a Gramáticas Formais
- Hierarquia de Chomsky – Wikipedia
- GeeksforGeeks – Chomsky Hierarchy
- Neso Academy – Chomsky Hierarchy (YouTube)

---

10. Questionário

1. **O que vem a ser um compilador?**
   Um **compilador** é um programa de software que atua como tradutor: converte um programa escrito em uma **linguagem de alto nível** (C, Java, Python) para uma representação de **baixo nível** (assembly, bytecode ou código de máquina). O compilador realiza essa conversão em múltiplas etapas (análise léxica, sintática, semântica, geração de código intermediário e final). Formalmente, pode ser modelado como uma **máquina de Turing**, pois é capaz de processar qualquer linguagem de programação dentro de seu domínio definido.

2. **O que é bootstrapping?**
   Bootstrapping é o processo de escrever um compilador em sua **própria linguagem-fonte**. Inicialmente, cria-se um compilador mínimo em outra linguagem (como assembly). Esse compilador mínimo então compila uma versão mais avançada escrita na própria linguagem, permitindo melhorias iterativas. O processo torna o compilador **auto-hospedado** e facilita

portabilidade, otimização e evolução. Exemplos clássicos incluem compiladores de C escritos em C e o GCC.

3. **Qual o papel do analisador léxico? O parser é um decisor sintático?**
   O **analisador léxico (lexer)** transforma a sequência de caracteres do código-fonte em **tokens**, eliminando espaços em branco e comentários. Cada token tem um significado léxico específico (palavra-chave, identificador, operador). Já o **analisador sintático (parser)** verifica se os tokens formam construções válidas de acordo com a **gramática livre de contexto** da linguagem. Pode-se dizer que o parser atua como um **decisor sintático**, pois aceita ou rejeita cadeias de tokens conforme a gramática definida.

4. **Qual a importância da tabela de símbolos?**
   A **tabela de símbolos** é uma estrutura de dados central para o compilador. Ela armazena informações sobre cada identificador usado no programa, incluindo:

5. Nome

6. Tipo de dado
7. Escopo (local, global)
8. Categoria (variável, função, constante)

9. Atributos adicionais (endereço, parâmetros, valores iniciais)
   O analisador semântico e o gerador de código consultam a tabela de símbolos constantemente. Ela garante consistência no uso de variáveis e funções, auxilia na verificação de tipos e facilita a alocação de memória.

10. **Como se classificam lexer, parser, semântico e gerador de código na hierarquia de Chomsky?**

11. **Lexer**: reconhece padrões regulares de tokens → **Tipo 3 (Linguagens Regulares)**.

12. **Parser**: valida sentenças segundo CFGs → **Tipo 2 (Gramáticas Livres de Contexto)**.
13. **Analisador Semântico**: aplica regras contextuais como compatibilidade de tipos → próximo de **Tipo 1 (Gramáticas Sensíveis ao Contexto)**.
14. **Gerador de Código**: mapeia para instruções de máquina, equivalente ao poder de uma máquina de Turing → **Tipo 0 (Recursivamente Enumeráveis)**.

Recursos para estudo

- *Dragon Book*, Cap. 1, 2, 3, 4 e 5
- compilers ">Bootstrapping (Compilers) – Wikipedia
- GeeksforGeeks – Compiler Design
- Neso Academy – Compiler Playlist

---

11. Referências

- Aho, Alfred; Lam, Monica; Sethi, Ravi; Ullman, Jeffrey. *Compiladores: Princípios, Técnicas e Ferramentas* (Dragon Book) – 2ª ed., Pearson, 2008.
- Kernighan, Brian; Ritchie, Dennis. *The C Programming Language* – 2ª ed., Prentice Hall, 1988.

- MIT OCW – 6.035 Computer Language Engineering (Compilers)
- Rice University – Keith Cooper Compiler Resources
- Lexical Analysis – Wikipedia
- Parsing – Wikipedia
- Bootstrapping (Compilers) – compilers ">Wikipedia
- Hierarquia de Chomsky – Wikipedia
- GeeksforGeeks – Compiler Design Tutorials
- Tutorialspoint – Compiler Design
- Flex Manual
- Bison Manual
- LLVM Language Reference
- Pro Git Book
- Makefile Tutorial
- Godbolt Compiler Explorer
- Neso Academy – Compiler Design Playlist (YouTube)

---

12. Conceitos Básicos de Linguagens Formais e Autômatos (LFA)

Como pré-requisito para compreender compiladores, a disciplina de **Linguagens Formais e Autômatos (LFA)** fornece a base matemática para análise de linguagens de programação e para o entendimento de como lexers, parsers e analisadores funcionam.

Conceitos fundamentais

- **Alfabeto (Σ)**: conjunto finito e não vazio de símbolos. Ex.: Σ = {a, b}.
- **Cadeia (string)**: sequência finita de símbolos de um alfabeto. Ex.: "abba" ∈ Σ*.
- **Cadeia vazia (ε)**: cadeia de comprimento zero.
- **Linguagem (L)**: subconjunto de Σ*. Ex.: L = {a^n b^n | n ≥ 0}.
- Fecho de Kleene (*): conjunto de todas as cadeias possíveis sobre Σ.

Classes de gramáticas (ligadas à Hierarquia de Chomsky)

- **Tipo 3 – Gramáticas Regulares**: descritas por expressões regulares, reconhecidas por **Autômatos Finitos (AFD/AFN)**. Base para o **lexer**.
- **Tipo 2 – Gramáticas Livres de Contexto (CFGs)**: reconhecidas por **Autômatos com Pilha (PDA)**. Base para o **parser**.
- **Tipo 1 – Gramáticas Sensíveis ao Contexto**: reconhecidas por **Máquinas Lineares Limitadas (LBA)**. Relacionam-se a verificações semânticas complexas.
- **Tipo 0 – Gramáticas Recursivamente Enumeráveis**: reconhecidas por **Máquinas de Turing**. Representam o poder computacional máximo (qualquer programa executável).

Autômatos

- **AFD (Autômato Finito Determinístico)**: cada estado tem no máximo uma transição para cada símbolo de entrada. Exemplo: reconhecer números binários múltiplos de 3.
- **AFN (Autômato Finito Não Determinístico)**: pode ter múltiplas transições possíveis para o mesmo símbolo.
- **PDA (Pushdown Automaton / Autômato com Pilha)**: estende AFs com uma pilha, permitindo reconhecer linguagens balanceadas, como parênteses.

- **Máquina de Turing (MT)**: modelo geral de computação, com fita infinita, capaz de simular qualquer algoritmo.

Exemplo prático

- Linguagem: cadeias balanceadas de parênteses.
- Gramática CFG: S → (S) | SS | ε
- Reconhecida por um PDA que empilha ao ler "(" e desempilha ao ler ")".

Relação com compiladores

- **Lexer**: implementa AFs (AFD/AFN) para reconhecer tokens definidos por regex.
- **Parser**: implementa PDAs para validar expressões e estruturas sintáticas.
- **Semântico e além**: requerem verificações contextuais (Tipo 1) e computação geral (Tipo 0).

Siglas importantes

- **Σ**: alfabeto.
- **ε**: cadeia vazia.
- **AFD**: Autômato Finito Determinístico.
- **AFN**: Autômato Finito Não Determinístico.
- **PDA**: Pushdown Automaton / Autômato com Pilha.
- **MT**: Máquina de Turing.
- **CFG**: Context-Free Grammar / Gramática Livre de Contexto.
- **LBA**: Linear Bounded Automaton / Máquina Linearmente Limitada.

Recursos para estudo

- Hopcroft, Motwani, Ullman – *Introduction to Automata Theory, Languages, and Computation*.
- *Dragon Book*, Cap. 2 – Formal Languages and Grammars.
- Neso Academy – Theory of Computation Playlist (YouTube)
- GeeksforGeeks – Automata Theory
- Tutorialspoint – Automata Theory