UNIC – Universidade de Cuiabá
Marcio Bezerra Cavalcanti Junior
2025

Síntese da Teoria da Computação: De Autômatos Finitos a Máquinas de Turing

Disciplina: Linguagens Formais e Autômatos
Professor: Felipe Douglas

Cuiabá – MT, novembro de 2025

Autômatos Finitos e Linguagens Regulares/Livres de Contexto

Esta unidade estabelece a base da hierarquia de Chomsky, começando com as linguagens mais simples (Regulares) e avançando para as Livres de Contexto, que são cruciais para a análise de linguagens de programação.

Seção 1: Transformações e Minimização de Autômatos Finitos

O foco aqui é a otimização e padronização de Autômatos Finitos (AF), os modelos computacionais mais simples, usados para reconhecer **Linguagens Regulares**.

Conversão de AFND para AFD:

- **Contexto:** Temos dois tipos principais de autômatos finitos: **Determinísticos (AFD)** e **Não Determinísticos (AFND)**.
- **AFD:** Para qualquer estado e qualquer símbolo de entrada, existe *exatamente uma* transição definida. São fáceis de implementar em software.
- **AFND:** Permite *múltiplas* transições para o mesmo símbolo a partir de um estado, ou transições "vazias" (transições-ε) sem consumir um símbolo. São mais fáceis de *projetar* e conceituar.
- **A Equivalência:** O ponto crucial é que, embora os AFNDs pareçam mais "poderosos" devido à sua flexibilidade, eles **não são**. Para todo AFND, existe um AFD equivalente que reconhece exatamente a mesma linguagem.
- **O Processo (Construção de Subconjuntos):** A conversão é feita através do "algoritmo de construção de subconjuntos". Cada estado no novo AFD corresponde a um *conjunto* de estados do AFND original. O estado inicial do AFD é o fecho-ε (conjunto de estados alcançáveis via transições-ε) do estado inicial do AFND. As transições do AFD são calculadas vendo para quais conjuntos de estados o AFND se moveria.

Minimização de AFD:

- **Objetivo:** Após converter para um AFD (ou se já tivermos um), ele pode ter estados redundantes. A minimização cria o AFD com o *menor número possível de estados* que ainda reconhece a mesma linguagem.
- **Por que fazer?** Eficiência. Um AFD menor significa um analisador léxico (scanner) mais rápido e que consome menos memória.
- **O Processo (Estados Distinguíveis):** O algoritmo mais comum funciona encontrando pares de estados que são "indistinguíveis".

- **Base:** Começa-se separando os estados em dois grupos: Finais e Não-Finais. Qualquer estado final é, por definição, distinguível de qualquer estado não-final.
- **Iteração:** Repetidamente, verificam-se pares de estados dentro do mesmo grupo. Se, para um mesmo símbolo de entrada, dois estados q_i e q_j transicionam para estados que *já* estão em grupos diferentes (ou seja, já foram distinguidos), então q_i e q_j também devem ser distinguidos.
- **Conclusão:** O processo termina quando nenhuma nova distinção pode ser feita. Todos os estados que permanecem no mesmo grupo são "equivalentes" e podem ser fundidos em um único estado no AFD minimizado.

Seção 2: Gramáticas Regulares (GR)

Definição: As Gramáticas Regulares (Tipo 3 na Hierarquia de Chomsky) são o mecanismo *gerativo* para as Linguagens Regulares. Enquanto os Autômatos Finitos são *reconhecedores*, as Gramáticas Regulares *produzem* as strings da linguagem.

Produção (Regras): Elas têm regras de produção muito restritas. Todas as regras devem seguir uma de duas formas (mas não misturar):

- **Linear à Direita:** $A \to aB$ ou $A \to a$ (onde A e B são não-terminais e a é um terminal).
- **Linear à Esquerda:** $A \to Ba$ ou $A \to a$.

Equivalência com AF: Este é o resultado fundamental da Unidade 3.

- **GR $\leftrightarrow$ AF:** As Gramáticas Regulares geram *exatamente* o mesmo conjunto de linguagens que os Autômatos Finitos reconhecem.
- É possível converter qualquer Gramática Regular em um AFND equivalente.
- É possível converter qualquer AFD em uma Gramática Regular equivalente.
- Isso unifica o conceito de "Linguagem Regular": é uma linguagem que pode ser (1) descrita por uma Expressão Regular, (2) reconhecida por um AFD, (3) reconhecida por um AFND, ou (4) gerada por uma Gramática Regular.

Seção 3: Linguagens Livres de Contexto (LLC)

Definição (LLC): Damos um salto em complexidade. As LLCs (Tipo 2 na Hierarquia de Chomsky) podem descrever estruturas que as Linguagens Regulares não conseguem, notavelmente o **aninhamento** e a **recursão**.

- *Exemplo clássico:* A linguagem $L = \{ a^n b^n \mid n \ge 0 \}$ (qualquer número de 'a's seguido pelo *mesmo* número de 'b's) não é regular, mas é livre de

contexto. Um AF não pode "contar" os 'a's para garantir que o número de 'b's seja o mesmo, pois ele não tem memória.

Gramáticas Livres de Contexto (GLC):

- **Definição:** São o mecanismo gerador das LLCs. Elas são a base para a definição da sintaxe da maioria das linguagens de programação.
- **Produção (Regras):** As regras são mais flexíveis que nas GRs. A forma é $A \to \alpha$, onde:
    - A é um *único* símbolo não-terminal.
    - α é uma sequência *qualquer* de terminais e/DOU não-terminais (incluindo a string vazia, ϵ).
- **Por que "Livre de Contexto"?** O nome vem do fato de que a regra $A \to \alpha$ pode ser aplicada *sempre* que o não-terminal A for encontrado, independentemente dos símbolos que vêm antes ou depois dele (o seu "contexto").

Autômatos com Pilha, Máquinas de Turing e Computabilidade

Esta unidade explora os modelos computacionais necessários para reconhecer linguagens mais complexas e, finalmente, questiona os limites do que pode ser computado.

Seção 1: Autômatos com Pilha (AP) e Análise Sintática

Autômatos com Pilha (AP):

- **O que é:** O reconhecedor para as Linguagens Livres de Contexto (LLC).
- **Funcionamento:** É, essencialmente, um **AFND + uma pilha (stack)**. A pilha fornece a memória que faltava aos AFs.
- A transição de estado não depende apenas do estado atual e do símbolo de entrada, mas também do símbolo no **topo da pilha**.
- Em uma transição, o AP pode:
    - Mudar de estado.
    - "Empilhar" (push) um símbolo na pilha.
    - "Desempilhar" (pop) o símbolo do topo.
- **Relação com GLC:** Assim como AF $\leftrightarrow$ GR, temos a equivalência fundamental: **AP $\leftrightarrow$ GLC**. Os Autômatos com

Pilha reconhecem *exatamente* o conjunto de linguagens geradas pelas Gramáticas Livres de Contexto. A pilha é o que permite ao AP "contar" e "lembrar" (ex: para $a^n b^n$, ele pode empilhar um 'X' para cada 'a' lido e desempilhar um 'X' para cada 'b' lido).

Análise Sintática (Parsing):

- **O que é:** O processo de usar uma GLC para validar uma string e descobrir sua estrutura lógica. É o que o compilador faz após a análise léxica.
- **Derivação:** O processo de aplicar as regras de uma GLC, começando pelo símbolo inicial, para *gerar* uma string de terminais. Uma derivação pode ser "mais à esquerda" (sempre expande o não-terminal mais à esquerda) ou "mais à direita".
- **Árvores Sintáticas (Parse Trees):** Uma representação gráfica de uma derivação. A raiz da árvore é o símbolo inicial, os nós internos são não-terminais e as folhas são os terminais. A árvore mostra a estrutura hierárquica e como a string foi construída.
    - **Ambiguidade:** Um problema sério em GLCs é a ambiguidade—quando uma mesma string pode ser gerada por *duas ou mais* árvores sintáticas diferentes. Isso é problemático para compiladores, pois a semântica (significado) geralmente depende da estrutura da árvore.
- **Forma Normal:** São formatos padronizados para GLCs que simplificam provas e a implementação de algoritmos de parsing. As duas mais famosas são:
    - **Forma Normal de Chomsky (FNC):** Todas as regras são da forma $A \to BC$ ou $A \to a$.
    - **Forma Normal de Greibach (FNG):** Todas as regras são da forma $A \to a\alpha$ (começam com um terminal).

Seção 2: Linguagens Sensíveis ao Contexto e Máquinas de Turing

Linguagens Sensíveis ao Contexto (LSC):

- **O que é:** O próximo nível na hierarquia (Tipo 1). São mais poderosas que as LLCs.
- *Exemplo clássico:* $L = \{ a^n b^n c^n \mid n \ge 0 \}$. Um AP não pode reconhecer isso, pois ele precisaria "contar" os 'b's e 'c's depois de já ter esvaziado a pilha usada para contar os 'a's.
- **Gramáticas Sensíveis ao Contexto (GSC):** As regras de produção *dependem* do contexto. A forma geral é $\alpha A \beta \to \alpha \gamma \beta$, o que significa que A só pode ser substituído por γ se estiver no contexto α e β. Uma definição alternativa mais simples é que as regras são

"não-contrativas" ($A \to \alpha$ onde $|\alpha| \ge |A|$), exceto pela regra inicial.

- **Autômatos Lineares Limitados (ALL / LBA):** O reconhecedor para LSCs. É um tipo especial de Máquina de Turing onde a fita é *limitada* ao tamanho da string de entrada.

Máquinas de Turing (MT):

- **Modelo:** O modelo de computação mais poderoso e central da Teoria da Computação. É o "computador" teórico definitivo.
- **Componentes:**
    - Uma **fita infinita**, dividida em células.
    - Uma **cabeça de leitura/escrita**, que pode ler e escrever símbolos na fita.
    - Uma **unidade de controle** (com estados finitos), que dita as ações.
- **Funcionamento:** Em cada passo, com base no estado atual e no símbolo lido da fita, a MT:
    - Escreve um novo símbolo na fita (substituindo o antigo).
    - Muda para um novo estado.
    - **Move a cabeça da fita para a Esquerda ou para a Direita.** (Esta é a grande diferença do AP, que só avançava).
- **Variações:** MTs com múltiplas fitas, MTs Não-Determinísticas (MTN). Surpreendentemente, todas essas variações são *equivalentes em poder computacional* à MT padrão de fita única (embora possam ser mais rápidas).
- **Capacidade:** Uma MT pode computar *qualquer coisa* que um computador moderno pode. Elas reconhecem as **Linguagens Recursivamente Enumeráveis** (Tipo 0), o topo da Hierarquia de Chomsky.

Seção 3: Decidibilidade e os Limites da Computação

Decidibilidade e Indecidibilidade:

- **Linguagem Decidível (ou Recursiva):** Existe uma Máquina de Turing que *sempre para* (halta) e responde "sim" (aceita) ou "não" (rejeita) para qualquer string de entrada.
- **Problema Decidível:** Um problema que pode ser modelado como uma linguagem decidível. Ex: "Este AFD aceita esta string?" é decidível.
- **Linguagem Indecidível:** Uma linguagem para a qual *não existe* uma MT que sempre para e dá uma resposta correta.

- **O Problema da Parada (Halting Problem):** O exemplo mais famoso de problema indecidível, provado por Alan Turing. O problema é: "Dado um programa (uma MT) e uma entrada, esse programa irá parar ou rodar para sempre?". Turing provou que é *impossível* criar um algoritmo universal que resolva este problema para todos os casos.
- **Redução:** A técnica usada para provar que novos problemas são indecidíveis. Se você pode "reduzir" o Problema da Parada a um novo problema P, isso significa que "se P pudesse ser resolvido, o Problema da Parada também poderia". Como sabemos que o Problema da Parada *não pode* ser resolvido, então P também não pode.

Teses de Church e Turing:

- **Contexto:** Nos anos 1930, vários matemáticos (Turing, Church, Kleene) criaram modelos diferentes de "computação" (Máquinas de Turing, Cálculo Lambda, Funções Recursivas).
- **A Descoberta:** Todos esses modelos, embora parecessem muito diferentes, foram provados ser *equivalentes* em poder computacional.
- **A Tese (Não é um teorema, é uma hipótese):** Afirma que "Tudo o que é *intuitivamente computável* por um algoritmo pode ser computado por uma Máquina de Turing."
- **Limites da Computação:** A tese define o que significa "ser computável". O Problema da Parada e outros problemas indecidíveis estão *além* desse limite.

Aplicações e Revisão:

- **Compiladores:** A aplicação mais direta desta teoria.
    - **Análise Léxica** (quebra de código em "tokens") é feita por **Autômatos Finitos (Linguagens Regulares)**.
    - **Análise Sintática** (verificação da estrutura gramatical, construção da árvore) é feita por **Autômatos com Pilha (Linguagens Livres de Contexto)**.
    - **Análise Semântica** (verificação de tipos, etc.) e geração de código já entram em domínios mais complexos, que exigem o poder das **Máquinas de Turing**.
- **Revisão Geral (Hierarquia de Chomsky):**
    - **Tipo 3 (Regular):** Reconhecida por **AF**. Gerada por **GR**. (Ex: a*b*)
    - **Tipo 2 (Livre de Contexto):** Reconhecida por **AP**. Gerada por **GLC**. (Ex: a^n b^n)

- **Tipo 1 (Sensível ao Contexto):** Reconhecida por **ALL**. Gerada por **GSC**. (Ex: a^n b^n c^n)
- **Tipo 0 (Recursivamente Enumerável):** Reconhecida por **MT**. Gerada por gramáticas irrestritas.