Enzo Hubner Moura de Andrade
Kaio Victor Marroque de Jesus

O Papel das Linguagens Formais na Construção de Linguagens de Programação

Serra
2025

Enzo Hubner Moura de Andrade
Kaio Victor Marroque de Jesus

O Papel das Linguagens Formais na Construção de Linguagens de Programação

Artigo apresentado como requisito parcial para obtenção de nota na disciplina.

Instituto Federal do Espírito Santo

Serra
2025

Resumo

Este artigo apresenta uma análise abrangente sobre a profunda e intrínseca relação entre **Linguagens Formais**, a teoria de **Autômatos** e o processo de construção de **Linguagens de Programação** modernas. O objetivo é discutir como os fundamentos da computação teórica não apenas definem a sintaxe, mas também garantem a confiabilidade e a segurança das ferramentas de software contemporâneas. Adotando uma metodologia de revisão teórica, o trabalho examina os conceitos da Hierarquia de Chomsky, os modelos computacionais (Autômatos Finitos, Autômatos com Pilha e Máquinas de Turing), e a função das Gramáticas Formais. A relevância do tema é evidenciada ao demonstrar que a teoria formal sustenta cada fase do processo de compilação, desde a análise léxica até a verificação de tipos e segurança em tempo de compilação, sendo essencial para o desenvolvimento de ferramentas robustas e eficientes.

Palavras-chave: Linguagens formais. Linguagens de programação. Compiladores. Autômatos. Gramáticas.

Abstract

This paper presents a comprehensive analysis of the profound and intrinsic relationship between **Formal Languages**, **Automata** theory, and the construction process of modern **Programming Languages**. The main objective is to discuss how the foundations of theoretical computer science not only define syntax but also ensure the reliability and security of contemporary software tools. Adopting a theoretical review methodology, the study examines concepts from the Chomsky Hierarchy, computational models (Finite Automata, Pushdown Automata, and Turing Machines), and the function of Formal Grammars. The relevance of the topic is highlighted by demonstrating that formal theory underpins every phase of the compilation process, from lexical analysis to type checking and compile-time security, proving essential for the development of robust and efficient tools.

Keywords: Formal languages. Programming languages. Compilers. Automata. Grammars.

Sumário

0.1 Introdução 5
0.2 Conceitos Fundamentais de Linguagens Formais 5
0.3 Classificação das Linguagens Formais 5
0.4 Autômatos e Máquinas Reconhecedoras 6
0.5 Gramáticas Formais 7
0.6 Relação entre Linguagens Formais e Compiladores 8
0.7 Influência nas Linguagens de Programação Modernas 10
0.8 Estudos de Caso e Exemplos de Aplicação 11
0.9 Desafios e Limitações 11
0.10 Considerações Finais 12
Referências 13

0.1 Introdução

A computação moderna, em sua essência, é a arte e a ciência de construir programas que transformam dados de entrada em resultados de saída de maneira previsível e eficiente. No centro desse processo estão as Linguagens de Programação, que servem como a ponte de comunicação entre a mente humana e o *hardware* da máquina. A escolha de um tema que relaciona linguagens formais e linguagens de programação é justificada pela necessidade de compreender a fundação teórica que confere a essas ferramentas sua precisão e confiabilidade.

O problema central de nossa pesquisa reside em **como os modelos teóricos das linguagens formais fornecem a estrutura matemática rigorosa para a definição, reconhecimento e processamento das linguagens de programação**. Sem esse formalismo, as regras sintáticas e semânticas seriam ambíguas, resultando em compiladores ineficientes, *bugs* indeterminados e *softwares* inseguros.

0.2 Conceitos Fundamentais de Linguagens Formais

O estudo das linguagens formais teve sua origem na computação teórica, notadamente a partir dos trabalhos de Noam Chomsky e Alan Turing, buscando fornecer uma base matemática rigorosa para definir e manipular linguagens.

Uma **Linguagem Formal** pode ser definida como um conjunto de cadeias (ou sequências finitas de símbolos) formadas a partir de um **Alfabeto** finito. O alfabeto (Σ) é o conjunto de símbolos básicos permitidos na linguagem (por exemplo, $\Sigma = \{0, 1\}$ para binário, ou o conjunto de caracteres ASCII). Uma cadeia é qualquer sequência finita de símbolos do alfabeto. A linguagem é, portanto, um subconjunto específico do conjunto de todas as cadeias possíveis (Σ^*).

A importância conceitual desses fundamentos reside na capacidade de descrever com precisão absoluta as regras de construção de qualquer linguagem. Para uma linguagem de programação, isso significa que a sintaxe — o que é considerado um programa "válido" — é definida por um conjunto de regras inequivocamente formais. Isso é crucial, pois permite que um programa (o compilador ou interpretador) possa determinar de forma algorítmica se outro programa (o código-fonte) está bem formado.

0.3 Classificação das Linguagens Formais

As linguagens formais não são homogêneas; elas são classificadas hierarquicamente de acordo com sua complexidade estrutural e o poder computacional necessário para reconhecê-las. Essa classificação é conhecida como a **Hierarquia de Chomsky**.

Tabela 1 – Hierarquia de Chomsky: Classes de Linguagens e seus Modelos de Reconhecimento

Classe da Linguagem	Modelo de Reconhecimento	Descrição da Capacidade
Tipo-3: Linguagens Regulares	Autômato Finito (AF)	Estruturas simples baseadas em repetição e concatenação.
Tipo-2: Livres de Contexto (GLC)	Autômato com Pilha (AP)	Capazes de representar estruturas aninhadas e recursivas (sintaxe da maioria das LPs).
Tipo-1: Sensíveis ao Contexto	Autômato Linearmente Limitado	Substituição de símbolos depende do contexto circundante.
Tipo-0: Recursivamente Enumeráveis	Máquina de Turing (MT)	A classe mais geral; simula qualquer algoritmo computável.

A apresentação dessas classes é fundamental para entender a estruturação de um compilador. Por exemplo, a fase de análise léxica é resolvida por linguagens regulares (Tipo-3), enquanto a fase de análise sintática é resolvida por linguagens livres de contexto (Tipo-2).

0.4 Autômatos e Máquinas Reconhecedoras

Os autômatos são os modelos computacionais abstratos que se correlacionam diretamente com as classes de linguagens formais, agindo como máquinas capazes de **reconhecer** se uma determinada cadeia pertence ou não a uma linguagem específica.

1. **Autômatos Finitos (AF):** São os mais simples, correspondentes às Linguagens Regulares (Tipo-3). Um AF é caracterizado por um número finito de estados e transições definidas pela leitura de símbolos de entrada. Eles são usados na análise léxica (a primeira fase da compilação) para reconhecer *tokens* como palavras-chave, identificadores, números e operadores. A expressão regular para identificar identificadores é um exemplo de Linguagem Regular que pode ser modelada por um Autômato Finito Determinístico (DFA).

2. **Autômatos com Pilha (AP):** Correspondem às Linguagens Livres de Contexto (Tipo-2). O AP adiciona uma memória de pilha ilimitada ao Autômato Finito. Essa

pilha permite que a máquina se lembre de contextos passados, o que é essencial para processar estruturas aninhadas, como verificar se para cada parêntese de abertura existe um parêntese de fechamento correspondente. Eles são a base teórica para os *parsers* usados na análise sintática.

3. **Máquinas de Turing (MT):** O modelo computacional mais poderoso, correspondente às Linguagens Recursivamente Enumeráveis (Tipo-0). A MT possui uma fita de memória infinita e é capaz de simular qualquer algoritmo que possa ser executado em um computador. Ela serve como a definição formal do que é "computável", mas é mais relevante para a teoria semântica e a determinação de propriedades indecidíveis.

0.5 Gramáticas Formais

Enquanto os autômatos são os *reconhecedores* de uma linguagem (a parte do compilador que verifica se o código está correto), as **Gramáticas Formais** são os *geradores* da linguagem. Elas definem o conjunto de regras de produção que especificam como as cadeias válidas na linguagem devem ser construídas.

As gramáticas também seguem a Hierarquia de Chomsky (Gramáticas de Tipo-3, Tipo-2, Tipo-1 e Tipo-0), com cada tipo sendo capaz de gerar a respectiva classe de linguagem formal.

A função crucial das gramáticas na definição de linguagens reside em sua capacidade de:

1. **Definir a Sintaxe:** A Gramática Livre de Contexto (GLC, Tipo-2) é a ferramenta primária para definir a sintaxe das linguagens de programação. As regras de produção descrevem como os componentes sintáticos (*statements*) podem ser combinados. Por exemplo, a sintaxe básica de uma expressão aritmética é definida por regras como $E \to E + T \mid T$.

2. **Gerar a Árvore Sintática (Parse Tree):** Um *parser* usa as regras da gramática para analisar uma cadeia de entrada e construir uma árvore sintática. Essa estrutura hierárquica é essencial porque ela não apenas verifica se o código está sintaticamente correto, mas também impõe a ordem correta de operações e fornece a estrutura de dados que será usada pelas fases posteriores do compilador, como a análise semântica e a geração de código.

As gramáticas garantem, portanto, que a estrutura lógica do código seja rigorosamente definida e facilmente manipulada pela máquina, sendo a espinha dorsal teórica da fase de análise sintática.

0.6 Relação entre Linguagens Formais e Compiladores

As linguagens formais constituem a base teórica que permite a construção de compiladores confiáveis e eficientes. Cada fase do processo de compilação depende diretamente de modelos formais, garantindo precisão na interpretação do código-fonte. A Figura 1 ilustra esse fluxo.

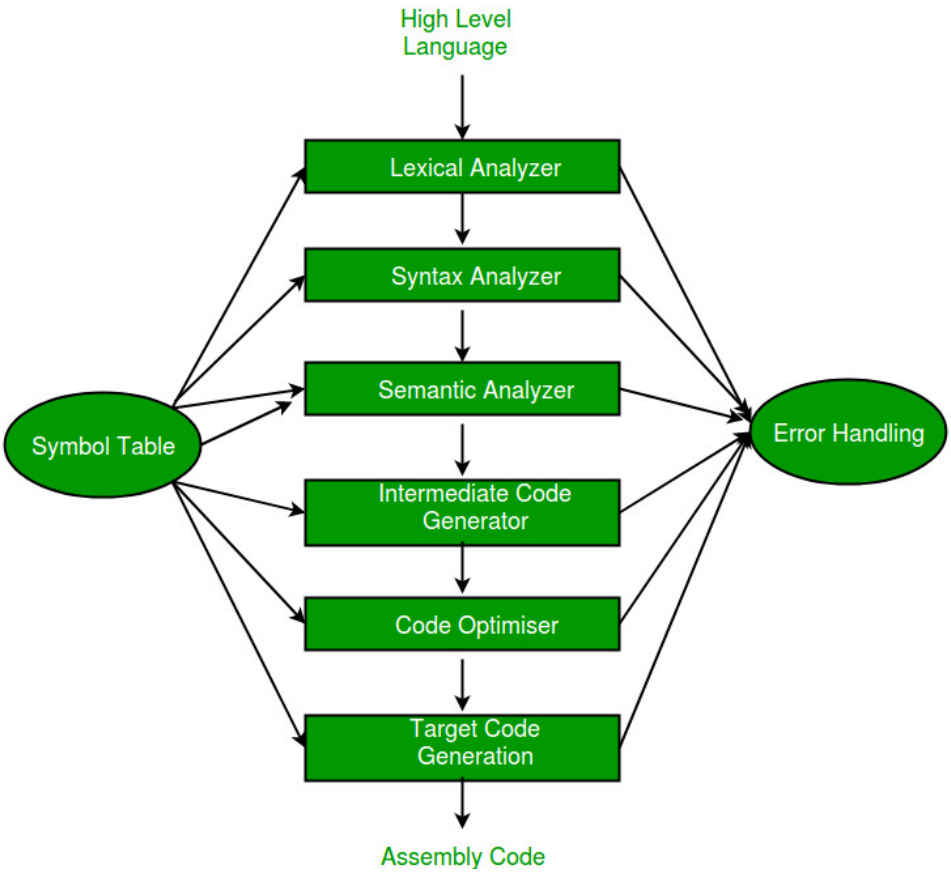


Figura 1 – Fluxo simplificado das etapas de compilação.

Análise Léxica

A análise léxica faz o reconhecimento de padrões usando **autômatos finitos**. Um exemplo simples é o reconhecimento de identificadores:

```
[a-zA-Z_][a-zA-Z0-9_]*
```

Essa expressão regular pode ser convertida para um autômato determinístico (DFA). A Figura 2 mostra um modelo típico.

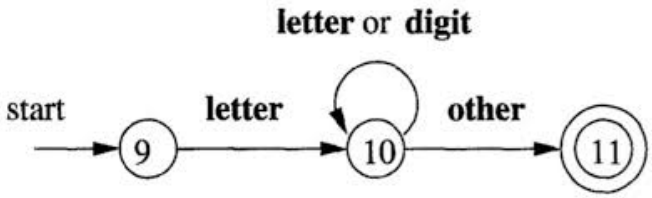


Figura 2 – Autômato finito determinístico para identificar identificadores.

Análise Sintática

A análise sintática utiliza regras de uma gramática livre de contexto (GLC). Por exemplo, um subconjunto de expressões aritméticas pode ser definido por:

```
E -> E + T | T
T -> T * F | F
F -> (E) | id
```

A partir dessa gramática, o compilador constrói uma *parse tree*, que pode se assemelhar à Figura 3.

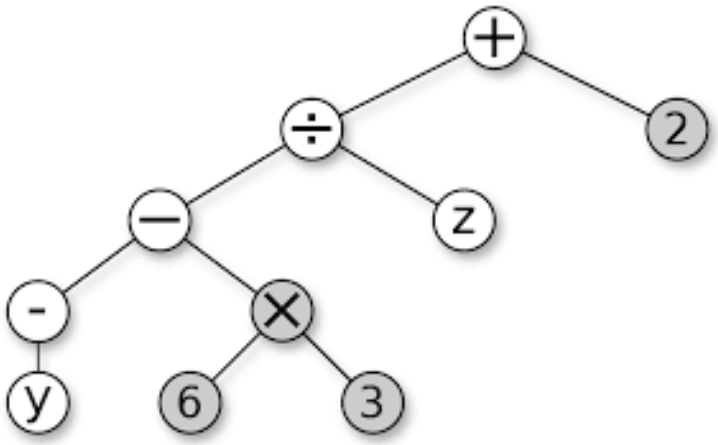


Figura 3 – Árvore sintática para a expressão `a + b * c`.

Análise Semântica

Nesta fase, o compilador verifica significado e contexto usando regras formais, como sistemas de tipos. Por exemplo:

```
int x;
x = "texto";   // Erro: tipos incompatíveis
```

Regras semânticas garantem consistência e previnem erros de tempo de execução.

—

0.7 Influência nas Linguagens de Programação Modernas

A influência das linguagens formais nas linguagens de programação contemporâneas é significativa e se manifesta desde a definição da sintaxe até a verificação automática de tipos e segurança.

Sintaxe e Estrutura

Linguagens como Python e Java têm suas sintaxes totalmente definidas por gramáticas formais. A Tabela 2 compara abordagens.

Linguagem	**Formalismo**	**Ferramenta de Parsing**
Python	GLC (PEG)	CPython Parser
Java	GLC	ANTLR / javac
Rust	GLC refinada	LALR Parser

Tabela 2 – Exemplos de linguagens e suas gramáticas formais.

Validação e Segurança

Ferramentas como analisadores estáticos e IDEs utilizam as regras formais da linguagem. Por exemplo:

- verificação de tipos em tempo de compilação (Java, Rust);
- *borrow checker* do Rust;
- validação de escopo lexical em Python.

Um exemplo em Rust:

```
let mut x = 10;
let y = &x;
x = 20; // Erro: y ainda referencia x
```

Esse comportamento decorre de regras formais de posse e empréstimo.

—

0.8 Estudos de Caso e Exemplos de Aplicação

Compilador de C (GNU GCC)

A linguagem C é definida por uma gramática formal que pode ser processada por ferramentas como *Bison*. Um trecho real simplificado da gramática:

```
function_definition:
    declaration_specifiers declarator compound_statement
```

Compiladores convertem esse tipo de regra em código que valida sintaxe e monta árvores abstratas.

JavaScript (ECMAScript)

O padrão ECMAScript especifica formalmente toda a linguagem. Motores como V8 seguem estritamente essa gramática, que contém regras como:

```
AdditiveExpression :
    MultiplicativeExpression
    AdditiveExpression + MultiplicativeExpression
```

Expressões Regulares (Regex)

Expressões regulares são exemplos diretos de linguagens formais na prática. Por exemplo:

```
^\d{3}\.\d{3}\.\d{3}\-\d{2}$
```

Esse padrão reconhece um CPF válido (estrutura), baseado em um autômato finito.

—

0.9 Desafios e Limitações

Apesar da utilidade das linguagens formais, diversos desafios surgem ao aplicá-las em linguagens modernas.

Simplicidade vs. Complexidade

Linguagens contemporâneas possuem sintaxes altamente complexas, frequentemente indo além das capacidades das gramáticas livres de contexto simples. Em alguns casos é necessário combinar:

- GLC + expressões regulares;
- heurísticas no parser;
- análises semânticas antecipadas para resolver ambiguidades.

Custo Computacional

Alguns algoritmos de análise sintática têm complexidade elevada, dificultando sua aplicação em compiladores que exigem alta performance.

Propriedades Indecidíveis

Problemas como determinação exata do tempo de execução ou detecção completa de loops infinitos são indecidíveis. Isso limita qualquer forma de análise semântica totalmente precisa.

Barreira Cognitiva

Os formalismos podem ser difíceis de aprender. Muitos estudantes e profissionais encontram dificuldade em compreender conceitos como derivação, fechamento e tabelas de parsing.

—

0.10 Considerações Finais

Este artigo apresentou uma análise abrangente sobre a relação entre linguagens formais, compiladores e linguagens de programação modernas. Foram discutidas aplicações práticas, exemplos reais e representações gráficas que evidenciam a importância desses fundamentos na computação.

Observou-se que conceitos formais sustentam desde etapas básicas, como a análise léxica, até elementos mais avançados, como segurança baseada em tipos. Apesar dos desafios, a teoria continua essencial para o desenvolvimento de ferramentas robustas, eficientes e seguras.

Como trabalhos futuros, destaca-se a criação de novos modelos híbridos de análise, a integração entre técnicas formais e inteligência artificial, e o estudo de linguagens cujo design seja totalmente guiado por fundamentos matemáticos rigorosos.

Referências

1 AHO, Alfred V. et al. **Compiladores: Princípios, Técnicas e Ferramentas**. 2. ed. São Paulo: Pearson Prentice Hall, 2008.

2 CHOMSKY, Noam. Three models for the description of language. **IRE Transactions on Information Theory**, v. 2, n. 3, p. 113–124, 1956. Disponível em: <https://doi.org/10.1109/TIT.1956.1056813>. Acesso em: 15 dez. 2025.

3 COOPER, Keith; TORCZON, Linda. **Construindo Compiladores**. 1. ed. Rio de Janeiro: Elsevier, 2013.

4 JUNG, Ralf et al. Stacked Borrows: An Aliasing Model for Rust. In: **Proceedings of the ACM on Programming Languages (PACMPL)**. Volume 4, número ICFP, p. 1–32, 2020. Disponível em: <https://plv.mpi-sws.org/rustbelt/stacked-borrows/paper.pdf>. Acesso em: 15 dez. 2025.

5 MENEZES, Paulo Blauth. **Linguagens Formais e Autômatos**. 4. ed. Porto Alegre: Bookman, 2010.