PROJETO PRÁTICO DE UM MINI-COMPILADOR: INTEGRAÇÃO ENTRE ANÁLISE LÉXICA, SINTÁTICA E GERAÇÃO DE CÓDIGO NA LINGUAGEM TINY

Christiano da Silva Vieira

Universidade de Cuiabá - UNIC
Curso de Engenharia da Computação

Cuiabá, 10 de Novembro de 2025

Sumário

RESUMO: Este trabalho descreve o desenvolvimento técnico e teórico de um mini-compilador para a linguagem didática *Tiny*. O projeto integra as fases fundamentais da compilação análise léxica, análise sintática e geração de código fundamentando-se nos princípios teóricos de linguagens formais e na estrutura de tradutores. A implementação, realizada em Python, utiliza uma arquitetura de máquina virtual baseada em pilha. Os resultados demonstram a viabilidade de implementações compactas para o ensino de engenharia de compiladores, comprovada através de testes automatizados.

Palavras-chave: Compiladores. Análise Léxica. Análise Sintática. Máquina Virtual. Linguagem Tiny.

ABSTRACT: This paper reports on the design and implementation of a small compiler for the educational language *Tiny*, covering lexical analysis, parsing, and code generation for a simple virtual machine. Grounded in formal language theory and compiler structure principles, the implementation in Python employs a stack-based architecture. Results demonstrate the feasibility of compact implementations for teaching compiler engineering, validated through automated testing.

Keywords: Compiler. Lexical Analysis. Parsing. Code Generation. Virtual Machine.

1 INTRODUÇÃO

A comunicação assertiva entre humanos e máquinas é o pilar central do desenvolvimento de software. Conforme Fedozzi (2018), a necessidade de criar soluções computáveis exigiu o estabelecimento de tradutores capazes de analisar a linguagem humana ou de alto nível e convertê-la em instruções inteligíveis para o computador. Esse tradutor complexo, denominado compilador, opera através de fases distintas e interconectadas que garantem a corretude e a eficiência do código final.

Apesar da complexidade inerente à construção de compiladores modernos, o ensino dessa disciplina beneficia-se enormemente de implementações compactas que isolam e evidenciam as fases clássicas do processo de tradução. O presente trabalho tem por objetivo detalhar a construção de um mini-compilador para a linguagem *Tiny*. O projeto visa não apenas produzir código executável para uma máquina virtual (VM) minimalista, mas também demonstrar, na prática, a integração entre o *lexer*, o *parser* e o gerador de código (*codegen*), aplicando os conceitos teóricos de análise e síntese.

2 FUNDAMENTAÇÃO TEÓRICA E ARQUITETURA

A arquitetura de um compilador é classicamente dividida em duas grandes fases: análise (*frontend*) e síntese (*backend*). A fase de análise verifica a correção gramatical do programa fonte, subdividindo-se em etapas léxica, sintática e semântica, enquanto a fase de síntese preocupa-se com a geração e otimização do código alvo.

2.1 Análise Léxica

A análise léxica constitui a primeira etapa do *frontend*. Sua função primordial é a leitura do fluxo de caracteres de entrada para a produção de uma sequência de componentes léxicos, conhecidos como *tokens*. Segundo a literatura técnica, essa fase lida com linguagens regulares, identificando elementos básicos como palavras-chave, operadores e identificadores.

No contexto da linguagem *Tiny*, o analisador léxico foi projetado para tokenizar números, identificadores, operadores matemáticos e delimitadores, descartando elementos não significativos como espaços em branco, preparando o terreno para a verificação estrutural.

2.2 Análise Sintática

Após a tokenização, ocorre a análise sintática. Esta etapa verifica se a sequência de *tokens* obedece às regras gramaticais da linguagem, estruturando-os hierarquicamente, geralmente em formato de árvore. A maioria das linguagens de programação modernas pertence à classe das Linguagens Livres de Contexto (LLC), permitindo o uso de gramáticas que suportam recursividade e aninhamento de estruturas.

Para o projeto *Tiny*, optou-se pela implementação de um *parser* recursivo descendente. Esta escolha alinha-se aos métodos *top-down*, onde a análise parte da raiz da gramática em direção às folhas, uma técnica intuitiva para implementação manual de gramáticas que não apresentam recursão à esquerda.

2.3 Geração de Código

A fase de síntese, ou *backend*, traduz a representação intermediária gerada pela análise em código executável. Diferente de compiladores que visam uma arquitetura de hardware específica (como x86 ou ARM), este projeto adota uma abordagem híbrida, gerando código para uma máquina virtual baseada em pilha (*stack-based*). Essa abordagem simplifica a geração de código intermediário, pois elimina a complexidade da alocação de registradores, focando na avaliação de expressões através de operações de *push* e *pop*.

3 METODOLOGIA E IMPLEMENTAÇÃO

O compilador foi desenvolvido utilizando a linguagem Python, priorizando a clareza do código e a modularidade. A escolha do Python, embora seja uma linguagem interpretada, permite uma prototipagem rápida dos conceitos de autômatos e estruturas de dados necessários para a tabela de símbolos e a árvore sintática abstrata (AST). O processo de desenvolvimento seguiu três módulos principais, descritos a seguir.

3.1 O Módulo Léxico (Lexer)

Implementado no arquivo **lexer.py**, este módulo é responsável por reconhecer os padrões da linguagem *Tiny*. Conforme os princípios de linguagens regulares, o lexer consome o código fonte e produz um fluxo de objetos *token*, cada um contendo seu tipo (ex: **NUMBER**, **PLUS**, **ID**) e seu valor literal. O tratamento de erros léxicos ocorre nesta fase, rejeitando caracteres não pertencentes ao alfabeto da linguagem.

3.2 O Analisador Sintático (Parser)

O arquivo **parser.py** implementa a lógica gramatical. Utilizando a técnica de descida recursiva, o parser consome os tokens e constrói uma AST. A estrutura da gramática lida com a precedência de operadores matemáticos onde multiplicação tem prioridade sobre adição garantindo que a árvore reflita a ordem correta de avaliação. As estruturas de controle suportadas incluem atribuições (**a = ...**), expressões aritméticas e comandos de saída (**print**).

3.3 Geração e Execução (Codegen e VM)

O módulo **codegen.py** percorre a AST gerada e emite *bytecode* para a máquina virtual. A VM, implementada em **vm.py**, simula uma CPU simples que opera sobre uma pilha de valores. Por exemplo, uma operação de soma desempilha dois valores, soma-os e empilha o resultado. A metodologia de verificação incluiu testes automatizados via biblioteca *pytest*, garantindo que para um conjunto de entradas conhecidas, a saída da VM fosse idêntica ao esperado.

4 RESULTADOS E DISCUSSÃO

Para validar a integração dos componentes, foi utilizado um programa exemplo na linguagem *Tiny* que envolve atribuição, precedência de operadores e impressão de resultados. O código fonte submetido ao compilador é apresentado abaixo:

```
# Exemplo de programa (program.tiny)

a = 2 + 3 * (4 + 1);

print(a);
```

O processo de compilação converteu a expressão matemática corretamente, respeitando a precedência dos parênteses e da multiplicação. O teste de integração, executado via **main.py**, processou o arquivo e a máquina virtual produziu a saída **17**, confirmando a correção funcional da cadeia de tradução.

Além dos testes locais, o ciclo de desenvolvimento foi automatizado utilizando Integração Contínua (CI) via GitHub Actions. Isso assegura a reprodutibilidade dos resultados e serve como evidência de funcionamento.

A análise dos resultados indica que a implementação priorizou com sucesso a clareza e a modularidade, facilitando o entendimento das fases do compilador. Contudo, nota-se limitações intencionais, como o suporte reduzido a recuperação de erros sintáticos avançados uma característica complexa onde o compilador tenta continuar a análise após encontrar um erro e a ausência de otimizações de código intermediário, tópicos que, segundo Fedozzi (2018), compõem a fase de síntese avançada.

5 CONCLUSÃO

O desenvolvimento do mini-compilador para a linguagem *Tiny* cumpriu o objetivo de demonstrar pragmaticamente a integração das fases léxica, sintática e de geração de código. Ao conectar a teoria das linguagens livres de contexto e máquinas de pilha com uma implementação funcional em Python, o projeto ilustra como estruturas abstratas se traduzem em software funcional.

Trabalhos futuros podem expandir esta implementação para incluir a geração de código nativo ou a introdução de uma tabela de símbolos mais robusta para verificação de tipos (análise semântica), aproximando ainda mais o projeto educacional de compiladores industriais.

REFERÊNCIAS

FEDOZZI, Regina. **Compiladores**. Londrina: Editora e Distribuidora Educacional S.A., 2018.

VIEIRA, CHRISTIANO. **Projeto Prático de um Mini-Compilador: Integração Léxica, Sintática e Geração de Código**, 2025.