

# “Vython: Desenvolvimento de um Interpretador para uma Linguagem Baseada em Python”

Davi Cavalcante, João Miguel, Pablo Abdon, Felipe Cruz

Centro Universitário do Estado do Pará (CESUPA), Belém, Brasil.

E-mails: [davi23070028, joao00000092, pablo23070029, felipe.cruz23070305] @aluno.cesupa.br

**Abstract**—Este artigo descreve o projeto, desenvolvimento e implementação de um interpretador para a Vython, uma linguagem de programação educacional derivada do Python. A principal inovação da Vython é a substituição da sintaxe baseada em indentação por delimitadores de bloco explícitos (chaves { }), visando combinar a clareza semântica do Python com a estrutura de controle de fluxo de linguagens como C e Java. Utilizando a ferramenta ANTLR para a geração do analisador léxico e sintático, e o padrão de projeto Visitor para a implementação da lógica semântica, foi construído um interpretador funcional. O processo validou a aplicação prática de conceitos fundamentais da teoria de compiladores, como análise gramatical e percurso de Árvores Sintáticas Abstratas (AST). O resultado é um ambiente de execução capaz de processar scripts Vython, demonstrando a viabilidade da abordagem e servindo como uma base para futuras explorações no design de linguagens.

**Index Terms**—Compiladores, Interpretadores, ANTLR, Padrão Visitor, Design de Linguagens, Python.

## I. INTRODUÇÃO

A disciplina de Compiladores constitui um pilar fundamental na ciência da computação, oferecendo uma visão detalhada sobre a estrutura, tradução e execução de linguagens de programação. No âmbito deste curso, o presente artigo descreve o desenvolvimento e a implementação de uma linguagem de programação derivada do Python, denominada Vython, e seu interpretador customizado.

A escolha do Python como base para esta derivação foi estratégica, motivada pela sua sintaxe clara, facilidade de leitura e ampla adoção na indústria. Esta familiaridade permitiu que o projeto se concentrasse nos desafios centrais da engenharia de compiladores e na semântica da execução, em vez de na complexidade da sintaxe de linguagens de mais baixo nível.

No entanto, para fins didáticos e de exploração de paradigmas, nossa linguagem apresenta modificações estruturais intencionais em relação ao Python padrão. A principal inovação inclui a adoção de chaves ({} ) como delimitadores explícitos para blocos de código (em vez da indentação), alinhando o controle de fluxo a um estilo de programação mais comum em linguagens imperativas como C e Java.

O processo de construção envolveu a definição formal da gramática da linguagem utilizando a ferramenta ANTLR

[1], [2], que gerou o *lexer* e o *parser* necessários. A fase subsequente consistiu na implementação de um *Visitor* customizado, que percorre a Árvore Sintática Abstrata (AST) para realizar a interpretação e execução das instruções.

O objetivo final deste trabalho é duplo: demonstrar a aplicabilidade prática dos conceitos teóricos de análise léxica, sintática e semântica, e apresentar um interpretador funcional capaz de processar e executar eficientemente um código com características híbridas, que combinam a expressividade do Python com elementos estruturais customizados.

## II. METODOLOGIA

O desenvolvimento do projeto foi estruturado em etapas que abrangem desde a especificação da linguagem até a sua execução, utilizando ferramentas e práticas modernas de engenharia de software.

### A. Ferramenta de Geração de Analisadores

Para a construção dos analisadores léxico e sintático, optou-se pela utilização da biblioteca ANTLR [1]. A escolha se justifica por sua robustez, capacidade de gerar analisadores para múltiplas linguagens de destino e por abstrair a complexidade da implementação manual dos algoritmos de análise, conforme descrito por Parr [2].

### B. Definição da Gramática

A gramática da linguagem foi especificada em arquivos com a extensão `.g4`. Seguindo uma abordagem modular para promover a clareza e a manutenibilidade, as regras foram segregadas em dois arquivos distintos:

- **Analisador Léxico (Lexer):** Contém as definições dos *tokens* da linguagem (palavras-chave como `if`, `else`, `while`, identificadores, operadores etc.).
- **Analisador Sintático (Parser):** Define as regras estruturais e a sintaxe da linguagem, estabelecendo como os tokens podem ser combinados para formar construções válidas, como atribuições, expressões e blocos de código.

### C. Ambiente de Desenvolvimento e Dependências

O código do *lexer* e do *parser* foi gerado pelo ANTLR para a linguagem de destino Python 3. Para garantir a

reprodutibilidade do projeto e um ambiente de desenvolvimento limpo, foi utilizado um ambiente virtual para o isolamento completo das dependências. A gestão deste ambiente e dos pacotes necessários, como a biblioteca `antlr4-python3-runtime`, foi realizada com o gerenciador de pacotes `uv` [3]. A adoção desta ferramenta justifica-se por sua alta performance e por unificar as funcionalidades de criação de ambientes e instalação de pacotes.

#### D. Implementação do Interpretador com o Padrão Visitor

Após a análise sintática, foi desenvolvido um interpretador customizado para percorrer a AST e executar o código-fonte. A implementação foi arquitetada utilizando o padrão de projeto *Visitor* [4]. Esse padrão permite separar a lógica de processamento (a semântica da linguagem) da estrutura de dados que a representa (a árvore sintática).

No contexto do ANTLR, isso se materializou na criação de uma classe *visitor* customizada, que herda de uma classe base gerada automaticamente a partir da gramática. Para cada regra sintática de interesse (e.g., atribuição, expressão aritmética, laço de repetição), o método `visit` correspondente foi sobrescrito para executar a ação semântica apropriada, como atualizar uma tabela de símbolos, calcular o valor de uma expressão ou controlar o fluxo de execução do programa.

#### E. Validação e Depuração

Para validar a corretude das fases de análise, foram criados mecanismos de verificação. Um script de teste foi utilizado como entrada para gerar a cadeia de tokens, permitindo a inspeção do resultado da análise léxica. Adicionalmente, desenvolveu-se um script em *shell* para automatizar a geração e a visualização gráfica da árvore de *parser*, o que se mostrou uma ferramenta fundamental para a depuração da gramática sintática.

#### F. Controle de Versão e Disponibilidade

Visando a colaboração e a reprodutibilidade, todo o código-fonte do projeto — incluindo os arquivos de gramática `.g4`, a implementação do interpretador e os scripts auxiliares — foi disponibilizado em um repositório na plataforma GitHub.

### III. RESULTADOS E DISCUSSÃO

A aplicação da metodologia descrita foi validada através de um script de teste que implementa um algoritmo de cálculo de fatorial, projetado para abranger múltiplas características da linguagem Vython, conforme exibido na Listagem 1.

```

1  # DECLARACOES
2  argumento = 0
3  fatorial = 0
4  flamengo = [0,1,2,3,4,5,6,7]
5
6  # ALGORITMO
7  argumento = input("Digite um numero: ")
8  fatorial = argumento

```

```

9
10 if argumento == 0: {
11     fatorial = 1
12 }
13
14 while argumento > 1: {
15     argumento = argumento - 1
16     fatorial = fatorial * argumento
17 }
18
19 print("Fatorial: ", fatorial)
20 print("Flamengo: ", flamengo[1])

```

Listing 1. Script de teste para cálculo de fatorial e demonstração de funcionalidades da linguagem Vython.

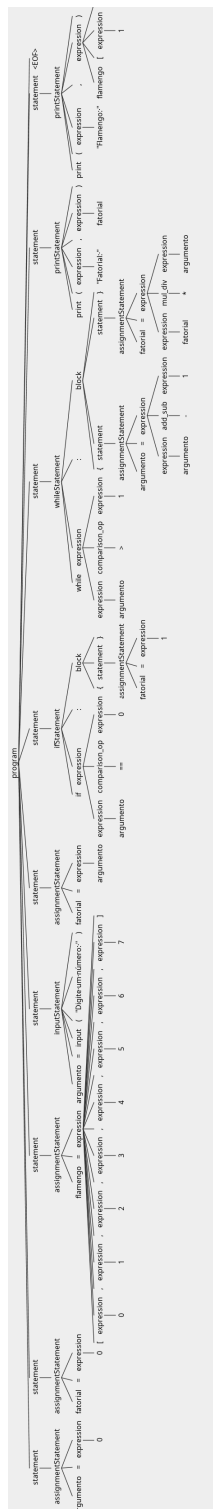
#### A. Validação do Analisador e do Interpretador

A validação do analisador léxico foi realizada com o comando `build.sh -tokens`, que processou o script da Listagem 1 e gerou uma cadeia de 77 tokens (incluindo o *EOF*). A saída demonstrou a correta identificação de todos os componentes léxicos, incluindo palavras-chave como `IF` e `WHILE`, operadores (`==`, `>`), e delimitadores específicos da Vython. A Tabela I na página a seguir exemplifica as principais classes de tokens e os lexemas correspondentes identificados a partir do script de teste.

Subsequentemente, a análise sintática foi verificada através da geração da árvore de *parser*. Utilizando os comandos de automação (`build.sh -gui` e `TestRig`), foi gerada a representação gráfica da estrutura sintática do código, conforme apresentado na Figura 1. A geração bem-sucedida de uma árvore de *parser* complexa como a da Figura 1 (também vista abaixo, após a tabela) é um resultado significativo. Ela comprova que a gramática da Vython é robusta o suficiente para analisar corretamente um algoritmo não trivial, reconhecendo a hierarquia das declarações, a estrutura do comando `if` e seu bloco, o aninhamento das operações dentro do laço `while`, e a sintaxe das chamadas de `print`, incluindo o acesso a um elemento de lista (`flamengo[1]`).

TABLE I  
EXEMPLOS DE LEXEMAS IDENTIFICADOS NO SCRIPT DE TESTE

Classe de Token	Lexema	Descrição
Identificadores	<code>argumento, fatorial, flamengo</code>	Nomes de variáveis definidas pelo usuário.
Constantes Inteiras	<code>0, 1, 2, 3, 4, 5, 6, 7</code>	Valores numéricos inteiros.
Strings	<code>"Digite um numero: ", "Flamengo: "</code>	Cadeias de caracteres delimitadas por aspas.
Palavras-Reservadas	<code>if, while</code>	Palavras-chave de controle de fluxo.
Operadores	<code>-, *, =, &gt;, ==</code>	Operadores aritméticos, de atribuição e lógicos.
Delimitadores	<code>( ) [ ] { } , :</code>	Símbolos de agrupamento e separação.
Funções Embutidas	<code>input, print</code>	Funções embutidas para captura e exibição de dados.

Fig. 1. Visualização panorâmica da árvore de *parser* gerada pelo ANTLR para o script da Listagem 1.

### B. Discussão sobre o Design da Linguagem

A decisão de substituir a indentação por chaves `{}` gerou um resultado interessante. Por um lado, conferiu à Vython uma estrutura de código explícita, eliminando a sensibilidade a espaços em branco, uma fonte comum de erros em Python. Isso alinhou a sintaxe a um paradigma mais tradicional de linguagens como C e Java. Por outro lado, essa escolha representa um afastamento consciente da filosofia “Pythônica” [5], que preza pela legibilidade por meio da indentação. O *trade-off* discutido é entre rigidez estrutural e clareza visual.

Um desafio notável na implementação do *Visitor* foi a gestão de escopo de variáveis. A solução envolveu a implementação de uma pilha de tabelas de símbolos, onde um novo escopo era criado ao entrar em um bloco (`{}`) e destruído ao sair (`}`), garantindo que as variáveis tivessem o tempo de vida correto. Este exercício prático reforçou a importância da análise semântica e da gestão de estado em tempo de execução.

Em suma, os resultados confirmam que a arquitetura escolhida (ANTLR + *Visitor*) é eficaz para a prototipagem e desenvolvimento de linguagens, especialmente para fins didáticos [6].

## IV. CONCLUSÃO

Este trabalho apresentou o projeto e a implementação da Vython, uma linguagem de programação interpretada que combina a simplicidade semântica do Python com uma estrutura de blocos baseada em chaves. Os objetivos definidos na introdução foram plenamente alcançados: demonstrou-se a aplicação prática dos conceitos teóricos da disciplina de Compiladores e entregou-se um interpretador funcional que valida o design da linguagem proposta.

A utilização da ferramenta ANTLR mostrou-se uma decisão acertada, acelerando o desenvolvimento ao automatizar a criação dos analisadores léxico e sintático. Da mesma forma, a adoção do padrão de projeto *Visitor* para o interpretador forneceu uma arquitetura modular e extensível, separando claramente as responsabilidades de análise sintática da execução semântica.

A principal contribuição reside na demonstração de ponta a ponta do ciclo de vida de uma linguagem de programação, desde sua concepção e especificação formal até a sua execução. O processo evidenciou os desafios práticos envolvidos, como a resolução de ambiguidades gramaticais e a gestão de escopo, consolidando o conhecimento teórico adquirido em aula. Em suma, o projeto Vython serve como um alicerce para futuras explorações no design de linguagens de programação.

## AGRADECIMENTOS

Agradecemos aos docentes e colegas do CESUPA pelo suporte técnico e acadêmico durante o desenvolvimento deste projeto.

## REFERENCES

- [1] ANTLR — ANother Tool for Language Recognition. [Online]. Available: <https://www.antlr.org/>
- [2] T. Parr, *The Definitive ANTLR 4 Reference*, 2nd ed. The Pragmatic Programmers, 2013.
- [3] Astral, “uv: An extremely fast Python package installer and resolver, written in Rust.” [Online]. Available: <https://github.com/astral-sh/uv>
- [4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [5] T. Peters, “The Zen of Python,” PEP 20. [Online]. Available: <https://peps.python.org/pep-0020/>
- [6] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Pearson, 2006.