ANHEMBI MORUMBI

BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

LARISSA SECARIO DA SILVA, 12522143765

DESENVOLVIMENTO DO COMPILADOR SNIPPY

SÃO PAULO

2025

LARISSA SECARIO DA SILVA

DESENVOLVIMENTO DO COMPILADOR SNIPPY

Projeto A3 da instituição Anhembi Morumbi com a proposta de desenvolver um compilador funcional para uma linguagem própria, contemplando as etapas de análise léxica, sintática, semântica e geração de código, conforme os requisitos da unidade curricular Teoria da Computação e Compiladores.

Orientador: Prof. W. Fernando

SÃO PAULO

2025

RESUMO

Este trabalho apresenta o desenvolvimento de um mini compilador chamado *Compilador SnipPy*, criado para a linguagem de programação SnipPy, que foi desenvolvida exclusivamente para fins acadêmicos e com o objetivo de demonstrar, na prática, os principais conceitos estudados na unidade curricular Teoria da Computação e Compiladores. A linguagem foi projetada de forma simples, suportando declarações de variáveis, operações aritméticas e lógicas, comandos de atribuição, estruturas condicionais, laços de repetição e instruções de entrada e saída. O compilador segue uma arquitetura modular composta pelas etapas de análise léxica, sintática e semântica, além da geração de código em C. O projeto também inclui a construção de uma Árvore Sintática Abstrata (AST) e um conjunto de testes destinados a demonstrar o funcionamento correto do sistema, incluindo a detecção de erros.

Palavras-chave: Compiladores, Linguagens de Programação, Análise Léxica, Análise Sintática e Análise Semântica.

SUMÁRIO

1 INTRODUÇÃO

Com o avanço das linguagens de programação e dos sistemas computacionais, surgiram ferramentas capazes de interpretar, traduzir e otimizar programas escritos por seres humanos. Entre essas ferramentas, os compiladores ocupam um papel importante, pois são responsáveis por transformar o código-fonte em representações intermediárias ou executáveis, criando uma ponte de comunicação entre o programador e a máquina.

Este trabalho apresenta o desenvolvimento de um mini compilador e de uma linguagem de programação própria. Essa linguagem, chamada SnipPy, foi criada exclusivamente para fins acadêmicos e é simplificada, inspirada em elementos de Python. Ela oferece estruturas básicas, como declaração de variáveis, operadores aritméticos e lógicos, comandos de atribuição, estruturas condicionais, laços de repetição e instruções de entrada e saída. O propósito da criação dessa linguagem é permitir a escrita de programas pequenos que possam ser analisados e traduzidos pelo compilador desenvolvido.

O compilador segue uma arquitetura composta por quatro etapas principais: a análise léxica, responsável por identificar os tokens; a análise sintática, que valida as regras gramaticais da linguagem; a análise semântica, que verifica significados, coerências e tipos; e, por fim, a geração de código, que converte o programa escrito em SnipPy para código C compilável e funcional.

2 OBJETIVO

O objetivo deste trabalho é desenvolver o *Compilador SnipPy*, responsável por interpretar e traduzir programas escritos na linguagem de programação SnipPy, criada exclusivamente para fins acadêmicos. O projeto busca aplicar, na prática, os conceitos estudados na unidade curricular Teoria da Computação e Compiladores, abrangendo as etapas de análise léxica, sintática, semântica e geração de código. Além disso, pretende-se demonstrar o funcionamento completo do processo de compilação, desde a leitura do código-fonte até a produção de um programa equivalente em linguagem C. Com isso, o trabalho reforça o entendimento dos principais componentes de um compilador e mostra como cada etapa contribui para a construção de um sistema funcional.

3 DESCRIÇÃO DA LINGUAGEM SNIPPY

A linguagem SnipPy foi criada como parte de um projeto acadêmico com o objetivo de oferecer uma linguagem simples para o estudo e o desenvolvimento dos conceitos envolvidos na construção de um compilador. Sua sintaxe é direta e foi inspirada na linguagem de programação Python.

SnipPy permite a escrita de programas compostos por declarações de variáveis, instruções de atribuição, operadores aritméticos e lógicos, estruturas condicionais, laços de repetição e comandos de entrada e saída. A linguagem adota o uso obrigatório do ponto e vírgula ao final de cada instrução simples.

Quanto aos tipos de dados, SnipPy oferece suporte a quatro categorias principais: int para inteiros, real para números de ponto flutuante, bool para valores lógicos e string para textos entre aspas. Esses tipos possibilitam a construção de expressões e estruturas de controle, além de permitir que o compilador execute verificações semânticas de forma adequada.

A linguagem conta com operadores aritméticos básicos (+, -, *, /, %), operadores relacionais (==, !=, <, <=, >, >=) e operadores lógicos (and, or, not),

possibilitando combinações expressivas em condições e cálculos. Estruturas como if/else, while e for permitem o controle de fluxo, enquanto print() e read() fornecem interações simples com o usuário.

SnipPy foi projetada para ser simples e com propósito totalmente acadêmico, permitindo compreender cada etapa de um compilador: o analisador léxico identifica os tokens, o analisador sintático verifica as regras gramaticais da linguagem, a análise semântica valida coerências e tipos, e a geração de código traduz o programa SnipPy para C. Assim, a linguagem cumpre seu papel como ferramenta educacional, possibilitando que o processo completo de compilação seja estudado de forma prática e objetiva.

4 GUIA DA LINGUAGEM SNIPPY

4.1 ESTRUTURA GERAL

SnipPy é uma linguagem criada para fins educacionais, com sintaxe inspirada em Python. Cada comando deve obrigatoriamente terminar com ponto e vírgula (;), exceto blocos (if, while, for). A Figura 1 apresenta um exemplo do código SnipPy.

Figura 1 - Código SnipPy

```
int x;
real y, z;

x = 10;
y = x + 3.14;

if (x > 5) {
    print(y);
}
```

4.2 TIPOS DE DADOS SUPORTADOS

A linguagem SnipPy trabalha com quatro tipos de dados básicos: int, real, bool e string. O tipo int representa valores inteiros, enquanto o tipo real é utilizado para números de ponto flutuante. O tipo bool representa valores lógicos, podendo assumir apenas as opções true ou false. Já o tipo string representa textos entre aspas.

Em relação aos valores que podem ser atribuídos a cada tipo, a Figura 2 apresentará exemplos dos literais aceitos pela linguagem, incluindo inteiros, números reais, valores booleanos e strings.

Figura 2 - Valores Aceitos em Cada Tipo

```
// Inteiro
int a, b;
a = 10;
b = -5;

// Real
real c,d;
c = 10.0;
d = -5.0;

// Boolean
bool e,f;
e = true;
f = false;


// String
string g,h;
g = "Ola";
h = "Teste 123";
```

4.3 DECLARAÇÃO DE VARIÁVEIS

Para declarar variáveis corretamente, sempre comece indicando o tipo, seguido pelo nome da variável e finalize a instrução com ponto e vírgula. Caso queira declarar mais de uma variável do mesmo tipo na mesma linha, separe os nomes usando vírgulas. Vale lembrar que, diferente de muitas linguagens, na linguagem SnipPy não é possível declarar uma variável e atribuir um valor a ela na mesma linha — esses passos devem ser feitos separadamente. A Figura 3 apresenta exemplos de declarações de variáveis corretas e incorretas.

Figura 3 - Exemplos de declarações corretas e incorretas

```
// Corretos
int a;
a = 10;

real b, c;
b = 2.0;
c = 1.0;

// Incorretos
int d = 10;
int e
```

4.4 ATRIBUIÇÕES

As regras semânticas de atribuição definem como valores podem ser associados a variáveis em um programa. Na linguagem SnipPy, para atribuir um valor a uma variável, é obrigatório que ela tenha sido declarada previamente. É importante destacar que, no caso de variáveis numéricas, não é permitido atribuir um valor real a uma variável inteira; entretanto, é permitido atribuir valores inteiros a variáveis do tipo real. Para variáveis do tipo string, somente valores do tipo string são aceitos, sem

formatação especial. Além disso, o uso de uma variável que não tenha sido declarada previamente resultará em erro. A Figura 4 apresentará um exemplo que ilustra essas regras de forma clara.

Figura 4 - Exemplos de atribuições corretas e incorretas

```
// Atribuições corretas
int a;
a = 10;

real b, c;
b = 15.0;
c = 15;

string d;
d = "Ola";


// Atribuições incorretas
int e;
string g;

e = 1.0;
g = "ola" + "mundo";
```

4.5 OPERAÇÕES PERMITIDAS

Na linguagem SnipPy, os operadores aritméticos (+, -, *, /, %) podem ser utilizados apenas com valores dos tipos *int* ou *real*, garantindo que as operações matemáticas sejam realizadas exclusivamente entre dados numéricos. No caso do operador de módulo (%), a operação funciona apenas com valores inteiros. Já os operadores relacionais (==, !=, <, <=, >, >=) comparam operandos numéricos e sempre produzem um resultado do tipo *bool*, indicando verdadeiro ou falso. Por fim, os operadores lógicos (and, or, not) funcionam somente com valores booleanos, sendo usados para combinar ou negar condições lógicas. A Figura 5 apresentará um exemplo ilustrando o uso correto desses operadores na linguagem.

Figura 5 - Exemplos de Operações

```
int a;
real b;

a = 10;
b = 5;

print(a+b);
print(a-b);
print(a*b);
print(a/b);

print(a==b);
print(a!=b);
print(a<b);
print(a<=b);
print(a>=b);
print(a>b);


// % Apenas pode ser usado com int
int c;
c = 10;
print(a/c);
```

4.6 ESTRUTURAS DE CONTROLE

Na linguagem SnipPy, as estruturas de controle permitem direcionar o fluxo de execução do programa de acordo com condições e repetições. A estrutura IF/ELSE segue o formato if (condicao) { ... } else { ... }, onde a condição deve obrigatoriamente ser do tipo *bool*, determinando qual bloco de comandos será executado. A estrutura WHILE utiliza o formato while (condicao) { ... } e executa repetidamente seu bloco enquanto a condição, também booleana, permanecer verdadeira. Já o comando FOR possui a forma for (i = 0; i < 5; i = i + 1) { ... } e a variável i deve ser declarada antes do FOR, permitindo controlar laços com inicialização, condição e incremento

definidos. A Figura 6 apresentará um exemplo que ilustra o funcionamento correto dessas estruturas de controle dentro da linguagem.

Figura 6 - Exemplos das estruturas de controle

```
// Exemplo com IF/Else
int a, b;
a = 10;
b = 20;

if (a>b) {
    print("A é maior que B");
} else {
    print("A é menor que B");
}


if (a == 10 and a < 20) {
    print(a);
} else {
    print(0);
}

// Exemplo while
int d;
d = 0;
while (d<3) {
    print(d);
    d = d + 1;
}

// Exemplo FOR
int i;
for (i = 0; i < 5; i = i + 1) {
    print(i);
}
```

4.7 ENTRADA E SAÍDA

Na linguagem SnipPy, os comandos de entrada e saída são fundamentais para exibir informações ao usuário e receber valores durante a execução do programa. O comando print permite mostrar mensagens, variáveis ou expressões. É obrigatório finalizar cada instrução com ponto e vírgula. Já o comando read é utilizado para ler um valor fornecido pelo usuário e armazená-lo em uma variável previamente

declarada. A Figura 7 apresentará um exemplo ilustrando o uso adequado desses comandos na linguagem.

Figura 7 - Exemplo do uso do read e print

```
int x, y;

read(x);
read(y);

print(x+y);
```

5 ARQUITETURA DO COMPILADOR

A arquitetura do compilador SnipPy foi desenvolvida seguindo o modelo tradicional usado na construção de compiladores, onde o funcionamento é dividido em etapas que acontecem de forma sequencial. Cada uma dessas etapas tem um papel específico e transforma o programa de entrada aos poucos, começando pelo código-fonte escrito pelo usuário até chegar na versão final gerada pelo compilador, que no caso do SnipPy é um programa equivalente em linguagem C.

A primeira fase é a análise léxica. Nela, o compilador lê o arquivo de entrada caractere por caractere e transforma tudo em uma sequência de tokens, que são unidades básicas de significado dentro do programa. Para fazer isso, é utilizado um sistema de leitura com buffers circulares, implementado na classe LeitorArquivosTexto, o que permite avançar e retroceder na leitura sempre que necessário. O componente que realmente classifica os lexemas é o SnipPyLexico, que identifica elementos como números, identificadores, palavras-chave, operadores, símbolos especiais e strings. Essa fase também ignora espaços em branco e comentários e é responsável por detectar caracteres inválidos, gerando erros léxicos

quando ocorre algo fora do padrão. No final, o analisador léxico produz uma sequência de tokens que será usada pelo parser.

A etapa seguinte é a análise sintática, feita pela classe SnipPyParser. Aqui, o compilador verifica se a sequência de tokens segue as regras da gramática da linguagem. Para isso, o parser usa a técnica de análise descendente recursiva, na qual cada regra é representada por um método que vai reconhecendo partes do programa. Conforme as estruturas válidas são identificadas, o parser constrói a AST (Árvore Sintática Abstrata). Essa árvore representa o programa de forma hierárquica, sem detalhes desnecessários da sintaxe real. Ela é formada por nós como ProgramNode, VarDeclNode, AssignNode, além das estruturas de controle IfNode, WhileNode e ForNode, e vários tipos de expressões. Caso algum trecho do código não esteja de acordo com a gramática, o parser interrompe o processo e gera um erro sintático.

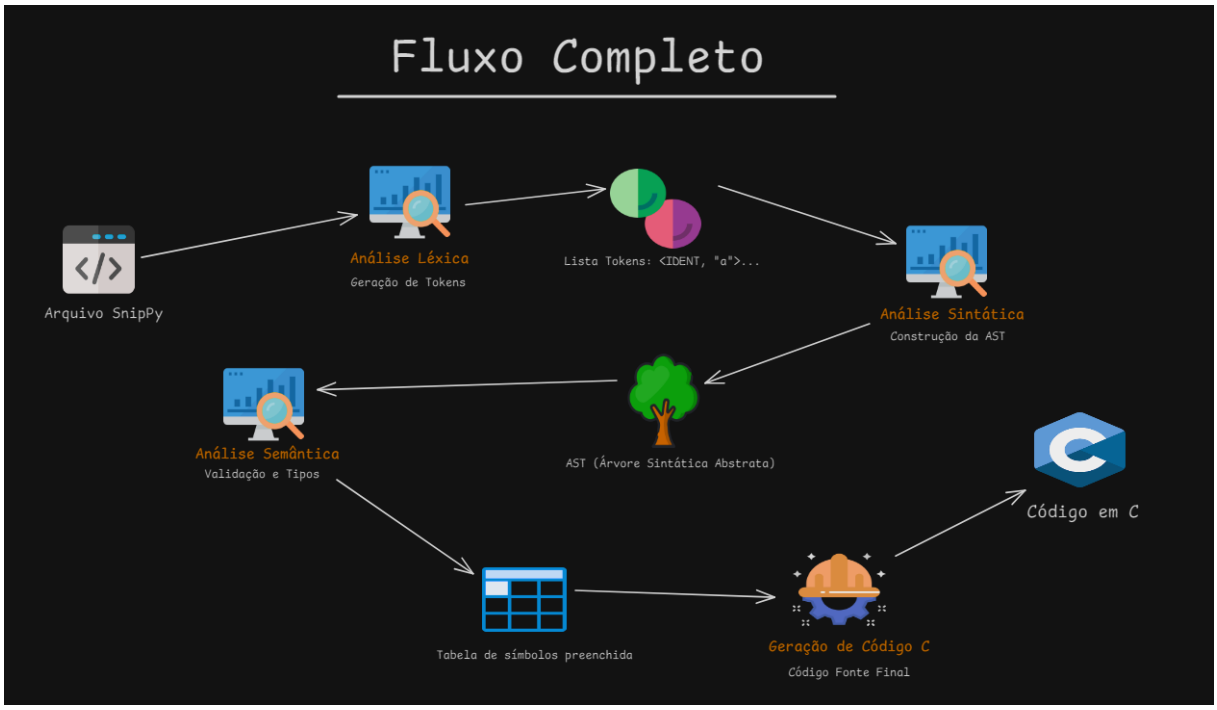
Com a AST pronta, o compilador passa para a análise semântica. Nessa etapa, o objetivo é verificar se o programa faz sentido em termos lógicos e de tipos. Quem executa essa fase é a classe SemanticAnalyzer, que usa uma tabela de símbolos para armazenar todas as variáveis declaradas e seus tipos. A análise semântica garante que variáveis só sejam usadas depois de declaradas, que não existam declarações duplicadas e que os tipos combinam com as operações realizadas. Também é nessa fase que se verifica, por exemplo, se as condições de comandos como if, while e for realmente são expressões booleanas. Caso alguma dessas regras seja violada, um erro semântico é lançado antes que seja gerado qualquer código.

Depois que o programa passa pelas verificações semânticas, o compilador chega à última etapa: a geração de código C. Essa tarefa é realizada pela classe CodeGeneratorC, que percorre a AST novamente e transforma cada nó em um trecho de código C equivalente. O objetivo é manter a lógica original do programa SnipPy, mas usando a sintaxe da linguagem C. Assim, declarações de variáveis, atribuições, condicionais, laços e comandos de entrada e saída são convertidos para suas versões em C. A função visit implementada para cada tipo de nó cuida de gerar o código correspondente, além de organizar a indentação e a estrutura dos blocos. Durante essa etapa, o gerador também trata detalhes como a escolha correta dos formatos no printf, diferenciando inteiros, reais, strings e valores booleanos, que são impressos

como "true" ou "false". No final, o compilador produz um arquivo C válido e pronto para ser compilado normalmente.

Na Figura 8, é possível visualizar de forma clara e resumida como todas essas etapas se conectam dentro do processo de compilação. O diagrama apresenta o fluxo completo, desde a leitura inicial do código-fonte até a geração final do programa em C, permitindo compreender de maneira mais visual cada etapa e suas relações.

Figura 8 - Fluxo Completo



6 EXEMPLOS DE CÓDIGO-FONTE E SAÍDA

Para ilustrar o funcionamento completo do compilador SnipPy, é possível observar um exemplo de código-fonte escrito na linguagem SnipPy e o resultado gerado no terminal, que corresponde ao código convertido para C. O código de entrada escrito em SnipPy será apresentado na Figura 9, enquanto a Figura 10 exibirá o código em C produzido pelo compilador.

Figura 9 - Código de Entrada Correta

```
// --- Declaração de variáveis ---
int x;
real y, z;
int contador;

// --- Atribuições ---
x = 10;
y = x + 3.14;
z = y * 2;

// --- Condicionais ---
if (x == 10 and y <= 20) {
    z = z / 2;
} else {
    z = 0;
}

if (true or false) {
    y = y - 1;
}

// --- Laços de repetição ---
while (x < 15) {
    x = x + 1;
}

for (contador = 0; contador < 5; contador = contador + 1) {
    print(contador);
}

// --- Entrada e saída ---
read(x);
print(y);
print(z);

// --- Strings ---
print("Ola, mundo!");
```

Figura 10 - Imagem do Terminal de Saída

```
[LEXICO] Token gerado: < PAREN_DIR, ) >
[LEXICO] Token gerado: < PONTO_VIRGULA, ; >
[LEXICO] Token gerado: < KW_PRINT, print >
[LEXICO] Token gerado: < PAREN_ESQ, ( >
[LEXICO] Token gerado: < IDENTIFICADOR, z >
[LEXICO] Token gerado: < PAREN_DIR, ) >
[LEXICO] Token gerado: < PONTO_VIRGULA, ; >
[LEXICO] Token gerado: < KW_PRINT, print >
[LEXICO] Token gerado: < KW_PRINT, print >
[LEXICO] Token gerado: < PAREN_ESQ, ( >
[LEXICO] Token gerado: < STRING, Ola, mundo! >
[LEXICO] Token gerado: < PAREN_DIR, ) >
[LEXICO] Token gerado: < PONTO_VIRGULA, ; >
[LEXICO] Token gerado: < EOF,  >

 ------ ANALISE SEMANTICA CONCLUIDA ------

 ------ CODIGO C GERADO ------

#include <stdio.h>

int main() {
    int x;
    double y, z;
    int contador;
    x = 10;
    y = (x + 3.14);
    z = (y * 2);
    if (((x == 10) && (y <= 20))) {
        z = (z / 2);
    }
    else {
        z = 0;
    }
    if ((1 || 0)) {
        y = (y - 1);
    }
    while ((x < 15)) {
        x = (x + 1);
    }
    for (contador = 0; (contador < 5); contador = (contador + 1)) {
        printf("%d\n", contador);
    }
    scanf("%d", &x);
    printf("%f\n", y);
    printf("%f\n", z);
    printf("%s\n", "Ola, mundo!");
    return 0;
}

 ================================== COMPILACAO CONCLUIDA COM SUCESSO ==================================
BUILD SUCCESSFUL (total time: 0 seconds)
```

As Figuras 11, 12 e 13 apresentarão exemplos de saídas incorretas referentes a cada uma das etapas.

Figura 11 - Erro Léxico

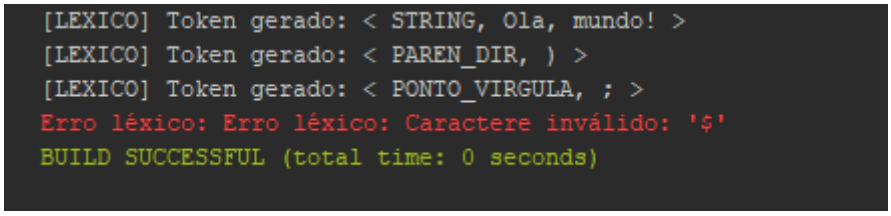


Figura 12 - Erro Sintático

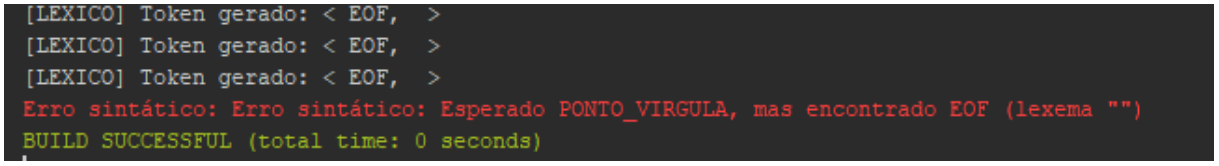
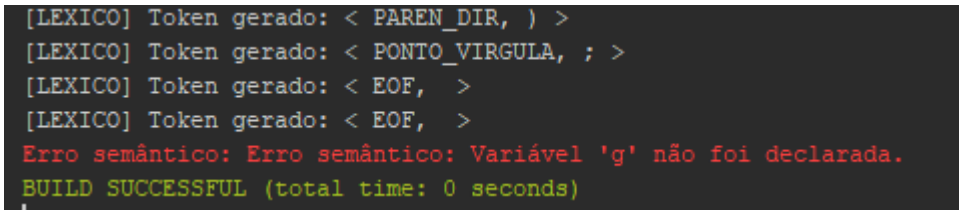


Figura 13 - Erro Semântico



7 DIVISÃO DE TAREFAS

Como o projeto foi desenvolvido individualmente, todas as etapas relacionadas ao compilador SnipPy foram realizadas pela mesma pessoa. Isso inclui tanto o planejamento da linguagem quanto a implementação prática de cada componente do compilador. Aluno responsável, Larissa Secario da Silva. RA, 12522143765

8 REPOSITORIO DO PROJETO

O repositório oficial do projeto está disponível no GitHub, no seguinte endereço: https://github.com/larissasecario/projeto-a3-compiladores.
Nele, é possível acessar todos os arquivos relacionados ao desenvolvimento do compilador, incluindo códigos-fonte, testes e documentação.

Dentro do repositório, a pasta docs contém um arquivo Markdown denominado Manual do Usuário, o qual apresenta instruções detalhadas sobre como instalar o projeto, configurar o ambiente e executar o compilador. Esse manual serve como guia principal para orientar o usuário desde a preparação inicial até a execução completa do sistema.

9 CONCLUSÃO

O desenvolvimento do compilador SnipPy permitiu compreender de forma prática como funciona o ciclo completo de tradução de uma linguagem de programação. A implementação das etapas léxica, sintática, semântica e da geração de código mostrou na prática como cada uma dessas fases tem um papel essencial para garantir que um programa seja interpretado corretamente e transformado em algo executável. Trabalhar na construção de cada módulo proporcionou uma experiência valiosa na aplicação de conceitos estudados em sala, como gramáticas, árvores sintáticas, regras semânticas e organização modular de software.

Além disso, o projeto evidenciou a importância do cuidado com detalhes, já que pequenas inconsistências em qualquer etapa podem comprometer todo o processo de compilação. Apesar dos desafios, o resultado final foi a criação de um compilador funcional e coerente, capaz de traduzir programas escritos em SnipPy para código C de forma adequada. A realização desse trabalho trouxe não apenas um entendimento mais profundo sobre compiladores, mas também contribuiu para o desenvolvimento de habilidades práticas em estruturação de código, depuração e análise crítica, consolidando a experiência como extremamente enriquecedora.

REFERÊNCIA

SPIVAK, Ruslan. *Let's Build a Simple Interpreter – Part 1.* Disponível em: https://ruslanspivak.com/lsbasi-part1/. Acesso em: 21 nov. 2025.

COMPILADORES PARA HUMANOS. *Structure of a Compiler.* Disponível em: https://johnidm.gitbooks.io/compiladores-para-humanos/content/part1/structure-of-a-compiler.html. Acesso em: 21 nov. 2025.

DELAMARO, Márcio. *Compiladores.* Disponível em: https://sites.icmc.usp.br/delamaro/slidescompiladores/compiladoresfinal.pdf. Acesso em: 21 nov. 2025.

UNIVERSIDADE PAULISTA – UNIP. *Compilador – Material de Apoio.* Disponível em: https://mediacdns3.ulife.com.br/PAT/Upload/5190218/compiladormaterialdeapoio_20251112180245.pdf. Acesso em: 21 nov. 2025.

UNIVERSITY OF NOTTINGHAM. *Compiler Design – Aula (vídeo).* YouTube, 2021. Disponível em: https://www.youtube.com/watch?v=NUCwfuS-V6s. Acesso em: 21 nov. 2025.

CURSO DE COMPILADORES – Playlist 1. YouTube. Disponível em: https://www.youtube.com/playlist?list=PLaPmgS59eMSEKNRIBxuBK4mJr-8pFP3lW. Acesso em: 21 nov. 2025.

COMPILADORES – Playlist 2. YouTube. Disponível em: https://www.youtube.com/playlist?list=PLRAdsfhKI4OWNOSfS7EUu5GRAVmze1t2y. Acesso em: 21 nov. 2025.

COMPILADORES – Playlist 3. YouTube. Disponível em: https://www.youtube.com/playlist?list=PLUDlas_Zy_qC7c5tCgTMYq2idyyT241qs. Acesso em: 21 nov. 2025.