

# Geração de Código da linguagem TPP

Henrique S. Marcuzzo<sup>1</sup>

<sup>1</sup>Departamento de Ciência da Computação  
– Universidade Tecnológica Federal do Paraná (UTFPR)

henriquemarcuzzo@alunos.utfpr.edu.br

**Abstract.** *This article describes the steps taken to perform the code generation work, with a brief explanation of the strategy adopted, details of how the code was produced, and a basic example output.*

**Resumo.** *Este artigo descreve os passos tomadas para a execução do trabalho de geração de código, com um breve explicação sobre a estratégia adotada, detalhes de como foi produzido o código e um exemplo básico de saída.*

## 1. Introdução

Este trabalho foi realizado com a linguagem de programação *Python* juntamente com todas as bibliotecas dos trabalhos anteriores mais as blibliotecas do *llvmlite* e *itertolls*, para que enfim seja possível gerar os códigos executáveis da linguagem de programação fictícia *.tpp*, durante o processo foi desenvolvido métodos para pegar o tipo da variável ou função no padrão LLVM, pegar variáveis na lista de variáveis já declaradas entre outros métodos para o processamento da árvore podada.

## 2. Implementação e geração de código

Para gerar o código intermediário *llvm* e por fim o código executável da linguagem *.tpp*, foi utilizado a árvore podada, e as tabelas de variáveis e função geradas na análise semântica. Para isso foram feitos métodos que identificassem variáveis globais e funções e as declarassem conformem fossem aparecendo na árvore.

O mesmo aconteceu para o corpo das funções que foi tratado como um sub-nó e processado utilizando a mesma lógica, mas agora, qualquer nó que fosse uma função, variável local, função leia ou escreva, repita, se, retorna ou atribuição fosse tratado como pertencente ao bloco da função atual.

Assim de maneira recursiva foi gerado todos os comandos intermediários *.ll* do *llvm*.

Foi desenvolvido também um código de entrada e saída na linguagem *C* que representaria as funções leia e escreva da linguagem *TPP*, assim na hora de gerar o executável, seria feita uma ligação entre o código de entrada e saída e o código que *.tpp*, funcionando como uma biblioteca padrão da linguagem.

## 3. Exemplo de entrada e saída

Para cada código *.tpp* submetido a está parte do projeto, será executada todas as partes anteriores do projeto e com os resultados gerados, a árvore começar a ser gerada como foi descrito na seção 2, isso irá gerar um código intermediário *.ll*, que será interligado com

o código da biblioteca padrão de entrada e saída, gerando um código *.bc*, para assim ser compilado pelo *clang* e gerar um executável *.o*. Com todas essas etapas concluídas com sucesso se torna possível executar o código.

Podemos ver um simples exemplo de saída quando executamos o código abaixo, que simplesmente lê 1 inteiro e 1 flutuante, guarda em suas respectivas variáveis e em seguida escreve os valores guardados:

```
inteiro_principal()
    inteiro: x
    flutuante: y

    x := 0
    y := 0.0

    leia(x)
    leia(y)
    escreva(x)
    escreva(y)

    retorna(0)
fim
```

Podemos observar o código intermediário *.ll* abaixo:

```
; ModuleID = "gencode-017.bc"
target triple = "x86_64-unknown-linux-gnu"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-"

declare void @"escrevaInteiro"(i32 %.1)

declare void @"escrevaFlutuante"(float %.1)

declare i32 @"leiaInteiro"()

declare float @"leiaFlutuante"()

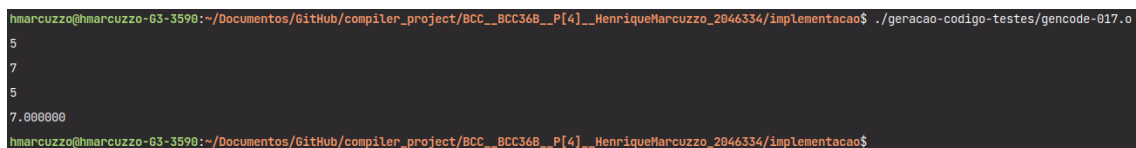
define i32 @"main"()
{
entry:
    %"x" = alloca i32, align 4
    %"y" = alloca float, align 4
    %"expression" = add i32 0, 0
    store i32 %"expression", i32* %"x"
    %"expression.1" = fadd float
0x0,          0x0
    store float %"expression.1", float* %"y"
    %"4" = call i32 @leiaInteiro()
```

```

store i32 %".4", i32* %"x", align 4
%".6" = call float @"leiaFlutuante"()
store float %".6", float* %"y", align 4
%".8" = load i32, i32* %"x"
call void @"escrevaInteiro"(i32 %".8")
%".10" = load float, float* %"y"
call void @"escrevaFlutuante"(float %".10")
br label %"exit"
exit:
ret i32 0
}

```

E por fim, o código executando pode ser visto na imagem 1.



```

hmarcuzzo@hmarcuzzo-63-3590:~/Documentos/GitHub/compiler_project/BCC__BCC36B__P[4]__HenriqueMarcuzzo_2046334/implementacao$ ./geracao-codigo-testes/gencode-017.o
5
7
5
7.000000
hmarcuzzo@hmarcuzzo-63-3590:~/Documentos/GitHub/compiler_project/BCC__BCC36B__P[4]__HenriqueMarcuzzo_2046334/implementacao$

```

**Figura 1. Saída da execução do código**

O código da biblioteca padrão da linguagem *.tpp* das funções de entrada e saída *leia* e *escreva* pode ser visto abaixo:

```

#include <stdio.h>

void escrevaInteiro(int ni) {
    printf("%d\n", ni);
}

void escrevaFlutuante(float nf) {
    printf("%f\n", nf);
}

int leiaInteiro() {
    int num;
    scanf("%d", &num);
    return num;
}

float leiaFlutuante() {
    float num;
    scanf("%f", &num);
    return num;
}

```

E seu código intermediário *.ll*, também pode ser observado abaixo:

```

; ModuleID = 'io.c'
source_filename = "io.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-
target triple = "x86_64-pc-linux-gnu"

@.str = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
@.str.1 = private unnamed_addr constant [4 x i8] c"%f\0A\00", align 1
@.str.2 = private unnamed_addr constant [3 x i8] c"%d\00", align 1
@.str.3 = private unnamed_addr constant [3 x i8] c"%f\00", align 1

; Function Attrs: noinline nounwind optnone uwtable
define dso_local void @escrevaInteiro(i32 %0) #0 {
    %2 = alloca i32, align 4
    store i32 %0, i32* %2, align 4
    %3 = load i32, i32* %2, align 4
    %4 = call i32 @i8*, ... @printf(i8* getelementptr inbounds ([4 x
    ret void
}

declare dso_local i32 @printf(i8*, ...) #1

; Function Attrs: noinline nounwind optnone uwtable
define dso_local void @escrevaFlutuante(float %0) #0 {
    %2 = alloca float, align 4
    store float %0, float* %2, align 4
    %3 = load float, float* %2, align 4
    %4 = fpext float %3 to double
    %5 = call i32 @i8*, ... @printf(i8* getelementptr inbounds ([4 x
    ret void
}

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @leiaInteiro() #0 {
    %1 = alloca i32, align 4
    %2 = call i32 @i8*, ... @__isoc99_scanf(i8* getelementptr inboun
    %3 = load i32, i32* %1, align 4
    ret i32 %3
}

declare dso_local i32 @__isoc99_scanf(i8*, ...) #1

; Function Attrs: noinline nounwind optnone uwtable
define dso_local float @leiaFlutuante() #0 {
    %1 = alloca float, align 4
    %2 = call i32 @i8*, ... @__isoc99_scanf(i8* getelementptr inboun
    %3 = load float, float* %1, align 4

```

```

    ret float %3
}

attributes #0 = { noinline nounwind optnone uwtable "correctly-rounded"
attributes #1 = { "correctly-rounded-divide-sqrt-fp-math"="false" "

!llvm.module.flags = !{!0}
!llvm.ident = !{!1}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"clang version 10.0.0-4ubuntu1 "}

```

#### 4. Referências

Moodle. Projeto de Implementação de um Compilador para a Linguagem T++. 2021. Disponível em: [https://moodle.utfpr.edu.br/pluginfile.php/183223/mod\\_resource/content/13/trabalho-03.md.notes.pdf](https://moodle.utfpr.edu.br/pluginfile.php/183223/mod_resource/content/13/trabalho-03.md.notes.pdf). Acesso em: 23 abr. 2021.

LLVM. LLVM Language Reference Manual. 2021. Disponível em: <https://llvm.org/docs/LangRef.html#add-instruction>. Acesso em: 15 mai. 2021.

GitHub. rogerioag/llvm-gencode-samples. 2021. Disponível em: <https://github.com/rogerioag/llvm-gencode-samples>. Acesso em: 15 mai. 2021.