# Compiladores - Geração de Código

**Alisson da S. Vieira**

[1]Ciência da Computação – Universidade Tecnológica Federal do Paraná (UTFPR)
Caixa Postal 271 – 87.301-899 – Campo Mourão – PR – Brasil

alissonv@alunos.utfpr.edu.br

***Abstract.*** *Code generation is the last step in the compilation process. Through the Abstract Syntax Tree generated in the semantic analysis, we generate the code of the input file. The work briefly addresses the methodology, implementation and examples.*

***Resumo.*** *A geração de código é o ultimo passo nos processos de compilação. Através da Árvore Sintática Abstrata gerada na Análise Semântica, geramos o código do arquivo de entrada. O trabalho aborda brevemente sobre a metodologia, implementação e exemplos.*

## 1. Introdução

A geração de código é a ultima parte do processo de compilação, onde se o código de entrada chegar aqui, quer dizer que o mesmo está apto a ser compilado, sem erros sitáticos e sem erros semânticos. Essa filtragem pelas análises posteriores é importante, para que apenas os códigos sem erros possam ser compilados. A análise visa transformar o código-fonte de entrada fornecido pelo usuário, em algo que é facilmente executada por uma máquina, como por exemplo, o código de máquina. Essa geração de código foi implementada com a linguagem de programação `Python`, em conjunto das bibliotecas: `Anytree`, `ply`, `pandas` e `llvmlite`. Com a ajuda da biblioteca `llvmlite`, conseguimos gerar códigos executáveis para a linguagem `T++`, que foi trabalhada ao longo das análises.

## 2. Implementação

Para a implementação, foi se utilizado as tabelas de variáveis, a de funções e a árvore sintática abstrata, todas elas sendo geradas na análise posterior: a analise semântica. O Código 1 é como foi gerado todo o processo.

**Código 1. Função que realiza o processo da geração do código**

```
1 def codeGenerator(file, root, dataPD, functionsPD, variablesPD):
2
3     declareVarGlobal(variablesPD)
4     declareFunctions(dataPD, functionsPD)
5
6     return modulo
```

Como vemos no Código 1, inicialmente é feita a declaração das variáveis globais, e logo em seguida, declaramos todas as funções. Foram feitas funções que geram tipos específicos de blocos, como: laço de repetição (na linguagem T++ é o “repita”) e laços de condições (na linguagem T++ é o “se”).

As funções de leitura e escrita para variáveis do tipo inteiro e flutuante foram implementadas na linguagem C e integradas ao código. Elas são utilizadas quando é identificado alguma chamada as funções “leia()”ou “escreva()”. Podemos ver como estão implementadas no Código 2.

**Código 2. Funções de leitura e escrita em C**

```c
#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;
}
```

Ao final de todo o processo, podemos ver o código gerado ao rodar o programa com o parâmetro “sm”. Vejamos o Código 3 como código de exemplo.

**Código 3. Código exemplo em T++**

```
inteiro e_dois(inteiro: n)
    se n = 2 entao
        retorna(1)
    fim

    retorna(0)
fim

principal()
    inteiro: i

    leia(i)

    escreva(e_dois(i))
fim
```

Este é um código simples, que irá ler um número escolhido pelo usuário, e irá retornar para o mesmo, o valor “0”caso esse número não seja o número “2”, e o valor “1”caso o número escolhido seja o número 2. Ao rodar o programa passando o argumento “sm”, podemos ver a saída gerada no Código 4.

**Código 4. Código gerado a partir do arquivo de entrada.**

```
; ModuleID = "modulo.bc"
target triple = "x86_64-unknown-linux-gnu"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80
    :128-n8:16:32:64-S128"

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

declare i32 @"leiaInteiro"()

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

declare float @"leiaFlutuante"()

define i32 @"e_dois"(i32 %".1")
{
entry:
  %"var_comp_if_r" = alloca i32
  %"var_comp_if_l" = alloca i32
  store i32 %".1", i32* %"var_comp_if_l", align 4
  store i32 2, i32* %"var_comp_if_r", align 4
  %".5" = load i32, i32* %"var_comp_if_l"
  %".6" = load i32, i32* %"var_comp_if_r"
  %"if_state" = icmp eq i32 %".5", %".6"
  br i1 %"if_state", label %"if_true", label %"if_end"
if_true:
  ret i32 1
if_end:
  ret i32 0
}

define void @"main"()
{
entry:
  %"i" = alloca i32, align 4
  %".2" = call i32 @"leiaInteiro"()
  store i32 %".2", i32* %"i", align 4
  %".4" = load i32, i32* %"i"
  %".5" = call i32 @"e_dois"(i32 %".4")
  call void @"escrevaInteiro"(i32 %".5")
  br label %"exit"
exit:
  ret void
}
```

Também foi implementado uma automatização referente a compilação desse código gerado, ao rodar o programa com o parâmetro “ar”, ele irá executar o Código 5.

**Código 5. Automatização do processo de compilação.**

```
def run(file):
    file = str(file)

    # comandos necessarios para gerar o codigo
    commands = [
        'clang -emit-llvm -S io.c',
```

```
7           'llc -march=x86-64 -filetype=obj io.ll -o io.o',
8           'llvm-link ' + file + '.ll io.ll -o ' + file + '.bc',
9           'clang ' + file + '.bc -o ' + file + '.o',
10          'rm ' + file + '.bc'
11      ]
12
13      # rodo os comandos
14      for command in commands:
15          subprocess.run(command.split(' '))
```

Após executádo, para testar o código o usuário apenas deverá rodar o seguinte comando no terminal: ".arquivo.tpp.o", e assim, poderá testar seu programa.

## 3. Exemplos

Para testar o programa, iremos utilizar o Código 6 de entrada do gerador. Esse sendo um código bem simples, onde temos uma variável global "a"e uma variável dentro do escopo da função "principal"nomeada "b". Inicialmente a variável "a"recebe o valor de "10", e logo após, a variável "b"recebe a variável "a", e por fim, retornamos a variável "b".

**Código 6. Arquivo de entrada: teste.tpp.**

```
1  inteiro: a
2
3  inteiro principal()
4    inteiro: b
5
6    a := 10
7
8    b := a
9
10     retorna(b)
11 fim
```

Passando os argumentos: "sm"e "ar", temos na saída o Código 7 sendo o código gerado do processo.

**Código 7. Código gerado a partir do arquivo de entrada.**

```
1  ; ModuleID = "modulo.bc"
2  target triple = "x86_64-unknown-linux-gnu"
3  target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80
       :128-n8:16:32:64-S128"
4
5  declare void @"escrevaInteiro"(i32 %".1")
6
7  declare i32 @"leiaInteiro"()
8
9  declare void @"escrevaFlutuante"(float %".1")
10
11 declare float @"leiaFlutuante"()
12
13 @"a" = common global i32 0, align 4
14 define i32 @"main"()
15 {
16 entry:
```

```
17    %"b" = alloca i32, align 4
18    store i32 10, i32* @"a"
19    %".3" = load i32, i32* @"a"
20    store i32 %".3", i32* %"b"
21    br label %"exit"
22  exit:
23    %".6" = load i32, i32* %"b"
24    ret i32 %".6"
25  }
```

Agora, basta rodar o comando: "./teste.tpp.o", em seguida o comando: "echo &?"para obter a saída mostrada na Figura 1.

```
als-v@DESKTOP-NRMU9ML:~/Compiladores/2. Compilador/4. Geração de Código$ ./geracao-codigo-testes/gencode-001.tpp.o
als-v@DESKTOP-NRMU9ML:~/Compiladores/2. Compilador/4. Geração de Código$ echo $?
10
```

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

## Referências


Gonçalves, R. A. Conjunto de testes. ttps://moodle.utfpr.edu.br/mod/resource/view.pp?id=138245.

Gonçalves, R. A. Slides - geração de código. ttps://moodle.utfpr.edu.br/pluginfile.pp/1112014/mod$_r esource/content/4/aula - 18 - geracao - de - codigo - ir - llvm - ir.md.slides.pdf$.

Lattner, C. Documentação llvmlite. ttps://llvmlite.readtedocs.io/en/latest/.