Implementação de um Compilador para a linguagem TPP: Geração de Código

Caio L. A. Miglioli

Departamento de Computação - Universidade Técnologica Federal do Paraná, Campus Campo Mourão.

caiomiglioli@gmail.com

Resumo. Este trabalho descreve a implementação da quarta parte do compilador para a linguagem TPP, a geração de código. Nesta parte traduzimos código em linguagem TPP em código de LLVM, que é em resumo, um backend de compilador. Para isso percorremos a árvore sintática gerada em passos anteriores em pré-ordem, executando funções de tratamento a cada nó referente à uma funcionalidade da linguagem. Utilizamos, por fim, o LLVM para compilar e gerar um código executável.

1. Introdução

O processo de implementação de um compilador para uma linguagem pode ser divida em quatro grandes partes: A análise léxica, a análise sintática, a análise semântica e a geração de código. Cada uma dessas partes possuem teorias complexas e distintas para que se resolvam os problemas levantados.

Na geração de código, buscamos transformar o código fonte inicial em código de máquina e gerar um arquivo executável, finalmente concluindo o processo de compilação. Mas para gerar o código, precisamos entender a hierarquia de todo o código fonte inicial, o que já está totalmente tratado na árvore gerada nas etapas anteriores.

Nesta etapa, traduzimos o código disponibilizado na árvore em código intermediário de LLVM, e utilizando a infra-estrutura do LLVM para traduzir tal código em um executável. Para fazer a tradução, utilizamos a api LLVM-Lite para python.

2. LLVM

O LLVM (Low Level Virtual Machine) é uma infra-estrutura de compilador capaz de gerar executáveis para os diferentes sistemas operacionais de forma semelhante à arquitetura Java. Mas no Java temos um conjunto fechado, enquanto podemos considerar o LLVM como um Backend para linguagens de alto nível.

No LLVM, temos um código assembly específico para LLVM (.ll) que gera um bytecode (.bc) que é traduzido em código de máquina (.o). Há diversos programas que traduzem diferentes linguagens de alto nível como C e C++ para .ll, e vice e versa. Nesta etapa utilizamos a api LLVM-Lite para traduzir nossa linguagem de alto nível para o assembly do LLVM (.ll).

2.1. LLVM Lite

A api LLVM Lite é uma biblioteca para Python que permite criar e ordenar instruções de LLVM gerando um arquivo .ll no final do processo. Essa api é baseada em módulos, que contém todo o código de um arquivo, e

em blocos, que correspondem a escopos como funções, if-else, e etc, e por fim, dentro desses blocos, inserimos as instruções desejadas.

```
Código Python:
# Variável inteira 'c'
c = builder.alloca(ir.IntType(32), name="c")
c.align = 4

# Armazena o valor inteiro 1 na variavel 'c'
num1 = ir.Constant(ir.IntType(32),1)
builder.store(num1, c)

-------------------------------------
Código resultante de LLVM:
%"c" = alloca i32, align 4
store i32 1, i32* %"c"
```

Código 1: Exemplo de Instruções do LLVM-Lite.

3. Implementação:

3.1. Lógica (função navigate())

Para implementar o código na ordem correta, foi utilizado a árvore podada resultante da análise semântica, pois sabemos que podemos acessar o código na ordem certa se percorrermos a árvore da esquerda para a direita, e podemos também acessar a precedência correta se percerrermos a árvore em pré-ordem.

Nessa implementação, percorremos a árvore em pré-ordem buscando os nós de tipo DECLARACAO_FUNCAO, DECLARACAO_VARIAVEIS, SIMBOLO (:=), RETORNA, LEIA, ESCREVA, CHAMADA_FUNCAO, SE e REPITA. Ao encontrar um desses nós, é chamado uma função para tratar a implementação da funcionalidade, e em seguida é retornado para o nó pai, ignorando os filhos (a fim de evitar redundâncias).

Na Figura 1, temos um exemplo do caminho percorrido na árvore. Na imagem, as setas em vermelho, mostram o caminho percorrido pelo loop principal, já as setas em verde mostram os nós encontrados que terão uma função de implementação executada a partir deles, e por fim, em azul, temos o caminho percorrido na sub-árvore 'corpo' executada pela função que implementa o nó 'DECLARACAO_FUNCAO'.

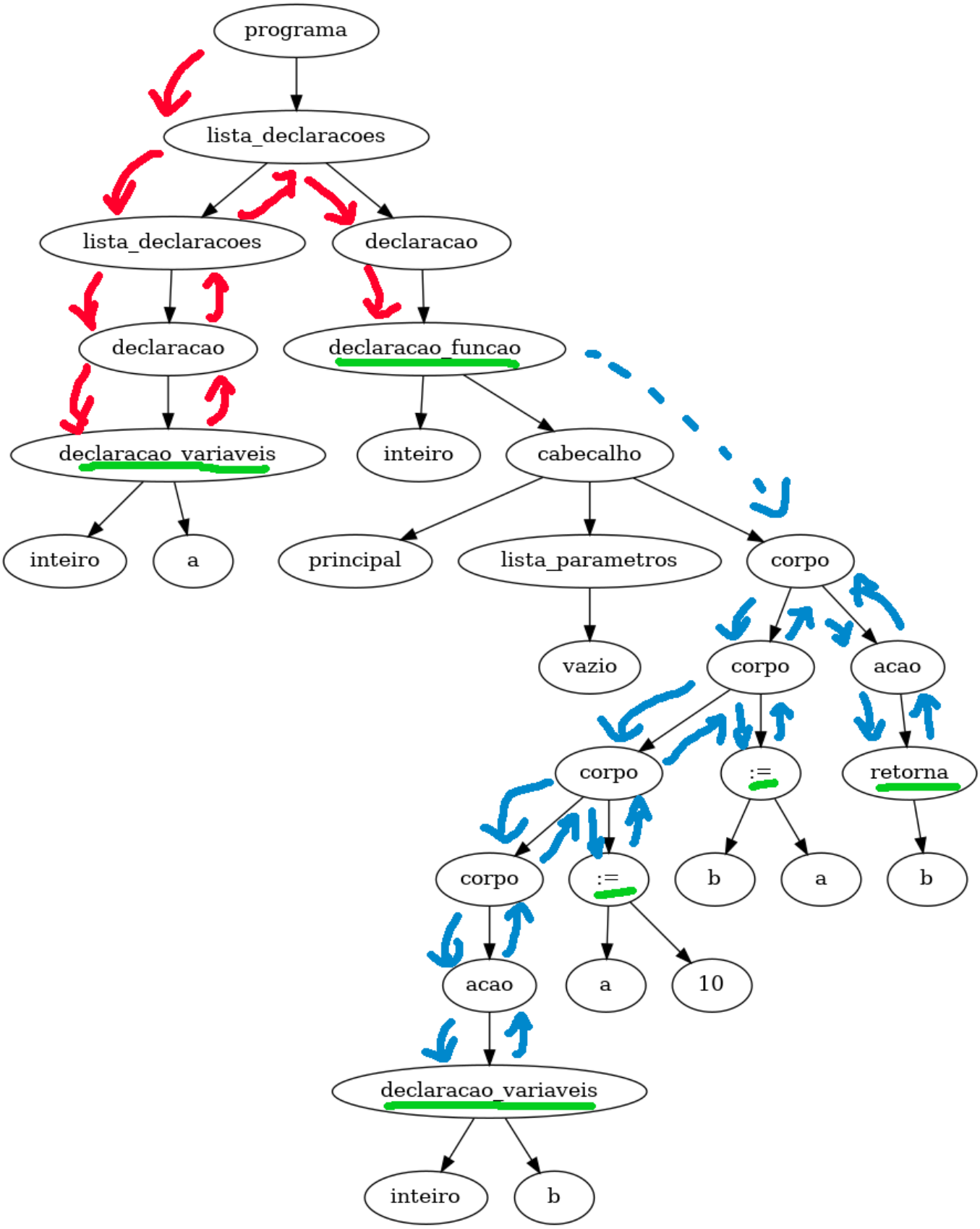

Figura 1: Exemplo de caminho em pré-ordem.

3.2. Funções de Implementação

3.2.1. Declaração de Função

Na declaração de função, utilizamos o apoio da tabela de funções gerada nas análises anteriores. Aqui iremos pegar as informações de tipo e parametros de forma mais eficiente. Com isso utilizamos as funções

do LLVMLite ir.FunctionType e ir.Function para gerar o cabeçalho.

Logo em seguida criamos os blocos de entrada e saída (esse ultimo terá a instrução de retorno). Logo em seguida, utilizando a função builder.alloca criamos uma variável de retorno e, para cada parametro, uma variável que irá conter o valor passado como argumento. Este último foi necessário pois o cabeçalho não cria uma variável que pode ser manipulada dentro da função.

Agora chamamos a mesma função que busca os nós e executa suas funções, mas ao invés de chamarmos com a árvore completa, utilizamos a sub-árvore 'corpo' correspondente ao corpo da função. Essa chamada (praticamente) recursiva vai se encarregar de produzir, no local correto, todas as instruções estipuladas no corpo.

Voltando do navigate, inserimos o pulo para o bloco de saída e no bloco de saída carregamos a variável de retorno com a função builder.load e retornamos esse valor utilizando a função builder.ret.

3.2.2. Declaração de Variável

Na declaração de variáveis, utilizamos o auxilio da tabela de variáveis gerada nas etapas anteriores, que além de ter informações úteis, utilizamos para guardar os ponteiros gerados pela alocação, que será utilizada em outras funções.

Logo em seguida verificamos se o tipo é inteiro ou flutuante, e também se o escopo é global ou local. Isso ditará se iremos utilizar a função ir.GlobalVariable ou builder.alloca.

Por fim, verificamos se há uma lista de variáveis, se é um vetor (mas que não foi implementado) ou se é uma variável única. Com isso fazemos o tratamento correto e chamamos a função correta como explicado anteriormente.

3.2.3. Atribuição

Na atribuição, temos um nó referente a variável a ser atribuida, e outro nó referente a expressão que resultada em um valor a ser guardado. Esta expressão deve ser resolvida primeira.

Tal expressão pode ser de quatro tipos: Constante, Chamada de Função, Expressão Aritmética ou Variável. Caso seja uma variável, apenas carregamos ela, caso seja uma chamada de função, tratamos ela com a mesma função que trata os nós 'CHAMADA_FUNCAO', mas caso ela seja uma expressão aritmética, temos que tratá-la separadamente.

Após termos o resultado da expressão, verificamos seu tipo através de uma simples expressão regular na string gerada pelo ponteiro, e executamos uma coerção utilizando os comandos builder.sitofp e builder.fptosi caso necessário.

Por fim, basta utilizarmos a função builder.store utilizando o ponteiro da variável a ser atribuida (ponteiro localizado na tabela de variáveis) e o ponteiro da expressão resultante.

3.2.3.1. Expressão Aritmética

As expressões aritméticas na árvore tem o formato de árvore binária, onde o nó pai é um símbolo referente a operação, enquanto os nós filhos são os valores à serem executados. Tais valores podem ser uma constante, uma variável, uma chamada de função, ou outra expressão aritmética.

Para cada um dos valores é executado uma checagem. Se o valor for uma constante, é retornado o valor, se for uma variável, é retornado o ponteiro de tal variável, se for uma chamada de função, é retornado a chamada da função (utilizando a função que trata estes casos), mas caso seja outra expressão aritmética, é chamado recursivamente esta mesma função implementada para lidar com o caso.

Por fim, com ambos os valores em mãos, é checado a expressão que deve ser efetuada e retornado o resultado de uma das funções builder.add, builder.sub, builder.mul ou builder.sdiv.

3.2.4. Retorna

A implementação do retorna é totalmente conectado à declaração de função. É aqui que é utilizado aquela variável de retorna criado junto de cada função.

Toda chamada retorna() é seguida de alguma expressão, mas como já existe uma função que lida com expressões, utilizamos ela aqui.

Após verificar o conteúdo da expressão e termos seu resultado em mãos, carregamos o ponteiro da variável 'retorna' armazenado na tabela de funções, e utilizamos o builder.store para armazenar seu valor. Por fim, utilizamos a função builder.branch com o bloco de saída e forçamos o programa a utilizar os comandos desse bloco definido na declaração de função.

3.2.5. Leia e Escreva

Há quatro funções geradas em um módulo separado: LeiaInteiro, LeiaFlutuante, EscrevaInteiro e EscrevaFlutuante. Todas implementadas em C e linkadas no processo de compilação.

Ao chamar uma dessas funções, precisamos identificar qual o tipo da variável ou expressão associada, e para isso foi criado uma função igual à 'verifyExpressionType' construida na etapa de Análise Semântica.

Após termos o tipo da variável a ser lida, ou o tipo da expressão a ser escrita, basta chamar a função correta utilizando o comando builder.call

3.2.6. Chamada de Função

A chamada de função em seus argumentos pode conter variáveis, valores, expressões e até outra chamada de função. Para encontrar os valores dos argumentos é preciso andar na árvore, buscado e tratando cada valor conforme necessário (utilizando as funções construídas anteriormente, e em caso de outra chamada de função, utilizando esta mesma de forma recursiva).

Após rodar a árvore e construir uma lista dos ponteiros referentes aos valores dos argumentos, basta utilizarmos a função builder.call com o nome da função e sua lista de argumentos, que automaticamente o LLVM-Lite irá construir a instrução completa.

3.2.7. Se-Então

Existem dois tipos de Se-Então, o que possui o 'Senão', e o que não possui. Para isso existe dois comandos, o builder.if_else e o builder.if_then. Na árvore, o nó 'SE' sempre possuirá o primeiro filho como a expressão condicional, e um ou dois filhos como corpo. Utilizando a quantidade de filhos, conseguimos definir qual comando utilizar.

Depois de criarmos o bloco 'if' e alocarmos no final da função, chamamos a função para tratar a condicional (a parte mais dificil dessa função) e utilizamos o comando mais apropriado para criar a instrução.

O LLVM-Lite possui, nesse caso, uma sintaxe muito agradável e direta de ser utilizado, o que, com a função navigate() sendo executada da mesma forma que na declaração de função, transforma isso em algo quase trivial.

```
cond = self.condicional(node.children[0])

with self.builder.if_else(cond) as (then, otherwise):
    with then:
        self.navigate(node.children[1])
    with otherwise:
        self.navigate(node.children[2])
```

Código 2: Parte da implementação do Se-Então.

3.2.7.1. Tratamento da Condicional

A condicional é definida por uma expressão lógica, que na nossa árvore sintática funciona da mesma maneira que uma expressão aritmética, onde o pai é o símbolo da operação, e os filhos são valores, variáveis, chamadas de função, expressões aritméticas ou expressões condicionais.

E de forma semelhante à expressão aritmética, aqui também tratamos os dois filhos e depois retornamos o ponteiro resultante da execução do comando. Aqui, porém, temos o comando builder.icmp_signed que executa as operações !=, ==, >, <, >= e <=, e os comandos builder.and_ e builder.or_ que executa as operações && e ||.

3.2.8. Repita

A funcionalidade de loop desenvolvida aqui foi a Repita-Até, que tem um comportamento que gerou a necessidade de um improviso. Os loops geralmente são executados enquanto a condicional tem valor verdadeiro, mas o Repita-Até é executado quando sua condicional tem valor falso, e para sua execução quando o seu valor é verdadeiro.

Como no LLVM-Lite não temos um comando como os de Se-Então para loops, foi utilizado o próprio builder.if_then em conjunto com o builder.branch para se gerar o loop. Mas como o Repita-Até executa apenas quando é falso, foi preciso inverter o resultado da condicional para que o código entrasse no bloco do if-then de forma correta. Isto pode ser visto no código 3.

```
#resolver expressao logica
cond = self.condicional(node.children[1])

#Inverter valor da expressao logica
cond = self.builder.not_(cond, name='Loop_Condicional$')

#retornar ao inicio do loop quando o valor invertido for verdadeiro
```

```
with self.builder.if_then(cond):
    self.builder.branch(loop_block)
```

Código 3: Parte da implementação do Repita-Até.

4. Exemplo de Uso

Para compilar um código TPP em um executável utilizando o compilador criado até aqui, é preciso ter todos os arquivos de código fonte do compilador (possuem o prefixo _) e mais o arquivo 'io.c'. Para compilar bast executar o comando python _tpp.py <Nome do arquivo tpp> <Nome do arquivo executavel a ser gerado>.

No código 4 podemos ver um exemplo simples contendo declaração de função, declaração de variável, atribuição de variável, repita-até, chamada de função, expressão aritmética, expressão lógica e retorna. No código 5 temos o código de assembly do LLVM (.ll), e no código 6 temos o resultado do código após ser compilado e executado.

```
inteiro principal()
    inteiro: ret
    ret := 0

    repita
        escreva(ret)
        ret := ret + 1
    até ret = 5

    retorna(0)
fim
```

Código 4: Código em TPP.

```
; ModuleID = "module.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 void @"escrevaFlutuante"(float %".1")

declare i32 @"leiaInteiro"()

declare float @"leiaFlutuante"()

define i32 @"main"()
{
entry:
  %"retorna$" = alloca i32, align 4
```

```
  %"ret" = alloca i32, align 4
  store i32 0, i32* %"ret"

  br label %"loop"
exit:
  %"ret_temp$" = load i32, i32* %"retorna$", align 4
  ret i32 %"ret_temp$"
loop:
  %".4" = load i32, i32* %"ret"
  call void @"escrevaInteiro"(i32 %".4")

  %".6" = load i32, i32* %"ret"
  %"sum$" = add i32 %".6", 1
  store i32 %"sum$", i32* %"ret"

  %".8" = load i32, i32* %"ret"
  %"igual$" = icmp eq i32 %".8", 5
  %"Loop_Condicional$" = xor i1 %"igual$", -1

  br i1 %"Loop_Condicional$", label %"loop.if", label %"loop.endif"

loop.if:
  br label %"loop"
loop.endif:
  store i32 0, i32* %"retorna$"
  br label %"exit"
}
```

Código 5: Código assebly do LLVM (.ll).

```
caio@caioubuntu:~/Desktop/Compiladores/Trabalho/Parte4/implementacao$
./geracao-codigo-testes/gencode-002.tpp.o
0
1
2
3
4
```

Código 6: Resultado da execução do código após compilado.

5. Conclusão

Não adianta analisar tudo se não produzir nada no final. A geração de código é o que traduz todo o processo intelectual que fizemos durante todo o processo em algo real, algo palpável, e a dificuldade, mesmo com um grande facilitador como o LLVM, mostra que criar um compilador pra mais simples linguagem é algo que demanda muito estudo e muito tempo.

Enfim geramos o nosso objetivo final: criar uma linguagem, algo que só um compilador funcional pode trazer à realidade. Desde a primeira etapa até a última, todo esse processo mostra como manter uma linguagem não é algo simples, e que, toda a facilidade criada para o usuário de alto nível se traduz em um caminhão de códigos e trabalho árduo para os desenvolvedores de mais baixo nível.

6. Referências

Louden, K. C. (2004). In Learning, C. editor, Compiladores: princípios e práticas. 1th edition.

Gonçalves, R. A. Slides do Professor.

Wikipédia. https://en.wikipedia.org/wiki/LLVM