Compiladores- Implementação e explicação sobre um compilador para a linguagem T++: Geração de código

Matheus Sapia Guerra[1]

[1]Universidade Técnologica Federal do Paraná - Campo Mourão (UTFPR)
Caixa Postal 271 – 87301-899 – Campo Mourão – PR – Brazil

guerramatheus2@gmail.com

Abstract. *This article aims to detail and explain an implementation of a compiler, more specifically the generation of code. For this, it is used libraries in Python that aims, facilitate and abstract in general, a construction of the compiler. The results obtained are successful from the implementation and relatively satisfactory, although not finalized, in fact generated the intermediate code and the restoration of the implementation and analogous. Thus, it is possible to conclude that despite complexity, it is possible to implement a compiler for an arbitrary language.*

Resumo. *Este artigo tem como objetivo detalhar e explicar sobre a implementação de um compilador, mais especificamente da geração de código. Para isto, é utilizado bibliotecas em Python que tem como finalidade facilitar e abstrair de maneira geral a construção do compilador. Os resultados obtidos a partir desta implementação é relativamente satisfatório pois apesar de não finalizado, de fato é gerado o código intermediário e o restante da implementação é análoga. Sendo assim, é possível concluir que apesar de complexidade, é possível implementar um compilador para uma linguagem arbitrária.*

1. Introdução

Linguagens de programação são criadas de acordo com necessidade de resolver algum tipo de problema especifico, facilitar o que já existe ou neste caso apenas como exercício. Porém a implementação de um compilador está longe de ser algo trivial.

A fase de geração de códigos é a ultima etapa de um compilador, onde dado um código de entrada o compilador retorna uma saída em LLVM-IR. Baseado nas fases anteriores da construção do compilador e na documentação das bibliotecas utilizadas, é possível realizar uma geração de códigos.

Este artigo tem como principal finalidade descrever como se implementa a ultima fase de um compilador (geração de código) para uma linguagem de programação arbitraria.

2. Fundamentação

Nesta seção é apresentado tópicos relevantes para o entendimento da geração de código, dentre esses tópicos podemos citar a o projeto LLVM e a biblioteca do Python *llvmlite*.

2.1. Projeto LLVM

Pode-se Dizer que o Projeto LLVM é uma coleção de tecnologia de compilador e ferramentas reutilizáveis, O nome LLVM não se refere exatamente a *Low Level Virtual Machine* e sim a o nome do projeto. O objetivo do LLVM é fornecer estratégia de compilação estática e dinâmica de linguagens de programação arbitrarias. Sendo assim o projeto LLVM é muito utilizado por vários projetos comerciais de código aberto e também em pesquisas acadêmicas.

Para este artigo é importante fundamentar um sub-projeto do LLVM que é o LLVM Core. O LLVM Core é um otimizador moderno que oferece suporte a várias arquiteturas diferentes, ou seja, é independente de origem e destino. Essas bibliotecas é construída através do LLVM IR, ou seja, isso possibilita a criação de uma linguagem de programação [LLV 2017]. A Figura 1 exemplifica o que foi explicado nesta subseção.

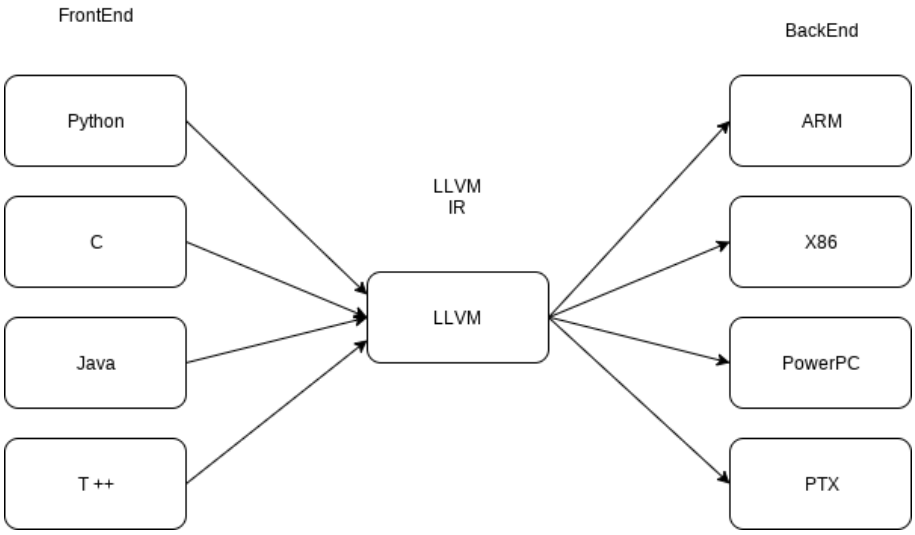

Figura 1. Exemplo de como funciona o LLVM-IR.

2.1.1. LLVMLite

O *llvmlite* é uma biblioteca para Python que tem como papel construir código intermediário, através da *intermediate representation* (IR). Sua implementação é em unicamente em Python[llv 2017]. A instalação dessa biblioteca pode ser feita utilizando o seguinte comando:

```
$ pip install llvmlite
```

3. Materiais

Para a etapa da geração de código foi utilizado a linguagem de programação Python. A biblioteca necessária pera realizar a geração é a *llvmlite* conforme mostrado na subseção 2.1.1. É importante ressaltar que é necessário importar também a implementação da Análise Semântica [Guerra 2017].

4. Implementação

Para a implementação da geração de código, foi utilizado as tabelas de simbolo e função geradas durante a análise semântica, o correto seria ter implementado a geração de código a partir de uma árvore sintática abstrata, porém esta árvore não foi implementada na análise semântica, sendo assim uma abordagem encontrada para contornar esta situação foi utilizar as tabelas de simbolo e função.

4.1. Classes

A geração tem apenas uma classe implementada, esta classe tem como finalidade percorrer a árvore sintática e com o auxilio das tabelas de simbolo/função, chamar funções que realizam a geração de código intermediário.

O primeiro módulo desta classe tem a finalidade de realizar declaração de variáveis globais, percorrendo a árvore até que encontre um nó chamado lista_declaracoes e que o seu neto seja exclusivamente uma declaracao_variaveis (para evitar que encontre variáveis dentro de funções), após isto é chamado uma função que obtém o tipo e o nome das variáveis. Sendo assim é possível chamar as funções que faz a geração de código para declaração de variáveis globais. O trecho a seguir demonstra como é feito a declaração de variáveis globais.

Código 1. Declaração de variáveis globais

```
1  var = ir.GlobalVariable(self.module, tipo, str(node.value))
2     if str(tipo) == 'i32':
3       zero = int(0)
4       var.initializer = ir.Constant(ir.IntType(32), zero)
5    else: se o tipo for float
6       zero = float(0)
7       var.initializer = ir.Constant(ir.FloatType(), zero)
8
9    var.linkage = "common"
10   var.align = 4
11   self.listaVar.append([var, escopo])
```

Dado o Código 1, a linha 1 faz a declaração de variável global passando por parâmetro o modulo, o tipo, e o nome da variável. A condicional das linhas 2 e 5 servem para verificar o tipo da variável, caso seja do tipo inteiro é atribuído o valor '0' caso contrario (será flutuante) é atribuído '0.0'. Por fim a tupla variável *llvmlite* e escopo é armazenada em uma lista para uso futuro.

Outro módulo implementado é o que monta funções, este percorre a árvore de forma análoga ao módulo apresentado anteriormente, mudando apensa as condicionais para entrar em determinados nós. A idéia deste módulo é procurar por declarações de função na arvore e utilizar a tabela de funções para chamar a função IR que gera o código para funções. O trecho de código a seguir mostra como declarar funções.

Código 2. Declaração de funções

```
1  if self.tabelaF[i].tipoRetorno == 'inteiro':
2     tipo_func = ir.FunctionType(ir.IntType(32), (args))
3    flag = 1
```

```
4 elif self.tabelaF[i].tipoRetorno == 'flutuante':
5   tipo_func = ir.FunctionType(ir.FloatType(), (args))
6   flag = 2
7 else:
8    tipo_func = ir.FunctionType(ir.VoidType(), (args))
9
10 func = ir.Function(self.module, tipo_func, name =
    self.tabelaF[i].nome)
```

O código 2 realiza verificação de tipos de retorno de uma função e os seus argumentos, tomando a linha 2 como exemplo, temos que para declarar o tipo da função é passado por parâmetro o tipo 'i32' e uma tupla de argumentos. Após esta verificação cria uma função passando por parâmetro o módulo, o tipo da função (definido anteriormente) e o nome da função.

Dentro deste módulo apresentado existem outras operações, como por exemplo, declaração de retorno, declaração de bloco de entrada e saída, declaração de variáveis, atribuição de valores para as variáveis (considerando que uma atribuição pode ser uma variável, um numero ou uma expressão). O processo de implementação é análogo aos apresentados anteriormente e as chamadas de função IR da biblioteca *llvmlite* podem ser encontradas na documentação [llv 2017] da biblioteca.

4.2. Execução

Para executar a geração de código, é necessário ter no mesmo diretório o *script* de análise léxica, análise sintática, análise semântica e por fim o *script* de geração de código, é necessário também um arquivo T++ para ser o código de entrada, atendendo estes requisitos o comando a seguir é um exemplo a ser utilizado para executar a compilação do código de entrada em T++.

```
1  $ python ger.py /geracao-codigo-testes/gencode-001.tpp
```

5. Resultados

Dado a implementação apresentada na 4, a saída esperada é um arquivo contendo o código intermediário de uma entrada de código T++. Em outras palavras, dado um código em T++ é realizado a análise léxica, análise sintática, análise semântica e caso o código esteja correto, é realizado a geração de código que tem como saída o código intermediário referente ao código T++. Tomando como exemplo a execução mostrada na subseção 4.2, temos o seguinte código de entrada:

Código 3. Código de exemplo em T++

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

```
9
10    retorna(0)
11 fim
```

A saída gerada para esta entrada é:

Código 4. Código intermediário gerado pelo compilador

```
1  ; ModuleID = "modulo.bc"
2  target triple = "x86_64-x86_64-linux-gnu"
3  target datalayout = ""
4
5  @"a" = common global i32 0, align 4
6  define i32 @"main"()
7  {
8  entrada_principal:
9    %"retorno" = alloca i32
10   store i32 0, i32* %"retorno"
11   %"b" = alloca i32, align 4
12   %"a" = load i32, i32* @"a"
13   store i32 %"a", i32* %"b"
14   br label %"saida_principal"
15 saida_principal:
16   %"retorno.1" = load i32, i32* %"retorno", align 4
17   ret i32 %"retorno.1"
18 }
```

6. Conclusões

Dado o exemplo apresentado na seção 5, podemos concluir que o compilador está de fato funcionando, pois gerou código LLVM IR, sendo assim o código gerado pode ser montado em qualquer arquitetura suportada pelo LLVM.

Podemos concluir também que a construção de um compilador é uma tarefa árdua e longa, requer um alto nível de habilidade em programação e também total domínio da linguagem utilizada.

Referências

(2017). LLVM. Disponível em http://www.llvm.org/ Acessado em 14/12/2017.

(2017). llvmlite. Disponível em http://llvmlite.pydata.org/en/latest/ Acessado em 14/12/2017.

Guerra, M. (2017). Compiladores: Implementação e explicação sobre um compilador para a linguagem t++: Análise semântica. Apresentado na disciplina de Compiladores.