Instituto Politécnico de Leiria
Escola Superior de Tecnologia e Gestão
Departamento de Engenharía Informática
Licenciatura em Engenharía Informática

COMO FUNCIONAM COMPILADORES

ESTUDANTE LUÍS JORGE MONTEIRO FERREIRA

Leiria, Julho de 2021

Instituto Politécnico de Leiria
Escola Superior de Tecnologia e Gestão
Departamento de Engenharía Informática
Licenciatura em Engenharía Informática

COMO FUNCIONAM COMPILADORES

DESDE O CÓDIGO-FONTE AO BINÁRIO

ESTUDANTE LUÍS JORGE MONTEIRO FERREIRA (2181140)

Leiria, Julho de 2021

RESUMO

Os compiladores são uma ferramenta importantissima de um programador e cada vez mais são uma ferramenta mais complexa e por vezes difícil de dissecar. Neste momento os compiladores são essenciais e indespensaveis para a vida de um programador e muitas das vezes os utilizadores deste tipo de ferramentas não olha para as várias vantagens e desvantagens que trazem e maior parte das vezes não percebem o seu comportamento, e por isso, podem fazer decisões erradas na hora de escolher uma linguagem de programação e a implementação do compilador associada a essa mesma linguagem.

Os objetivos deste documento é explicar brevemente o que é um compilador e a sua relevância, enumera os tipos de código usados e criados por este e os vários tipos de compiladores existentes, explicando os seus diferentes conceitos e dando alguns exemplos para complementar o seus casos de uso. Mais detalhadamente, o presente documento explica também as mais comuns fases de compilação de um compilador tradicional e explica brevemente cada fase.

Palavras-chave: Compiladores, tipos de compiladores, fases de compilação, código-fonte, código de máquina

A B S T R A C T

Compilers are a very important tool for a programmer and are increasingly a more complex and sometimes difficult tool to dissect. Nowadays, compilers are essential and indispensable for a programmer's life and many times users of these types of tools do not look at the various advantages and disadvantages they bring and most of the times do not understand their behavior, and therefore, they can make wrong decisions when choosing a programming language and the compiler implementation associated with that language.

The objectives of this document are to briefly explain what a compiler is and its relevance, enumerate the types of code used and created by it and the various types of existing compilers, explaining their different concepts and giving some examples to complement their cases of use. In more detail, this document also explains the most common compilation phases of a traditional compiler and briefly explains each phase.

Keywords: Compilers, compiler types, compilation phases, Source-code, Machine code

ÍNDICE

LISTA DE FIGURAS

LISTA DE TABELAS

LIST OF LISTINGS

ACRÓNIMOS

AOT	Ahead of Time.
ARM	Acorn RISC Machine.
ASIC	Application-specific Integrated Circuit.
AST	Abstract Syntax Tree.
AVR	Alf and Vegard's RISC.
BNF	Backus-Naur Form.
CISC	Complex Instruction Set Computer.
CPLD	Complex Programmable Logic Device.
CPU	Central Processing Unit.
DFA	Deterministic Finite Automaton.
DMD	Digital Mars D.
FPGA	Field Programmable Gate Array.
GC	Garbage Collector.
GCC	GNU Compiler Collection.
GHDL	G Hardware Design Language.
GNU	GNU's not UNIX.
HDL	Hardware Description Language.
IR	Intermediate Representation.

JIT	Just in Time.
JVM	Java Virtual Machine.
LLVM	Low Level Virtual Machine.
MIPS	Microprocessor without Interlocked Pipeline Stages.
NASM	Netwide Assembler.
OOP	Object Oriented Programming.
RISC	Reduced Instruction Set Computer.
VHDL	VHSIC Hardware Description Language.
VHSIC	Very High Speed Integrated Circuits.

1 INTRODUÇÃO

Com certeza que o aparecimento dos compiladores foi um grande passo para a evolução do software como hoje em dia o conhecemos. Em muitas áreas da programação esta ferramenta foi e ainda é essencial para a criação de vários tipos de software. Historicamente, os primeiros compiladores não são como os conhecemos hoje, porque muitos deles começaram as suas implementações em Assembly, complicando ainda mais todo o processo de construção dos mesmos. Como é de se calcular, este processo não é nada fácil, mesmo hoje em dia. Este documento pretende transmitir uma breve explicação sobre o processo de construção de um compilador comum para elucidar o conceito e assim perceber o melhor funcionamento desta ferramenta. (Parsons, 2003)

1.1 OBJETIVOS

1. **Perceber a definição de compilador:** Para decifrar o modo de funcionamento de um compilador é necessário primeiro entender conceptualmente o que é um compilador.
2. **Distinguir os vários tipos de compiladores:** Tipos de compiladores diferentes têm propósitos e modos de funcionamento diferentes. É necessário distinguir, por exemplo, o que difere de um compilador que compila para código de máquina para um compilador de bytecode.
3. **Entender sucintamente as várias fases de compilação:** É fulcral entender sucintamente como internamente as várias fases de compilação se processam, bem como a sua ordem e o porquê.

1.2 ESTRUTURA DO RELATÓRIO

No capítulo 2 os conceitos abordados ao longo deste documento são brevemente explicados. Conceitos como a definição de um compilador, os vários tipos de código

usados num compilador e os vários tipos de compiladores. Estes conceitos são sempre que possível acompanhados de um exemplo prático para os melhor correlacionar.

No capítulo 3 é abordado de forma simplificada cada fase de compilação de um compilador típico. Estas fases estão divididas entre o *frontend* do compilador e o *backend*. Sobre o *frontend* é abordado a fase de pré processamento, *lexing*, *parsing*, análise semântica e *intermediate code generation*. Sobre o *backend* é abordado o *optimizer* e o *target code generation*.

No capítulo 4 é abordado uma pequena conclusão, sumarizando os pontos focados neste documento e um conjunto de conceitos futuros a ter em conta.

2

CONCEITOS

O modo de funcionamento de um compilador pode ser visto como um tema bastante abstrato. Para perceber melhor este tema é necessário primeiro consolidar alguns conceitos base abordados ao longo deste documento.

2.1 COMPILADOR

Conceptualmente, um compilador é um software que permite converter um determinado tipo de código-fonte de alto nível em outro objeto de código, normalmente código de baixo nível. Tipicamente este tipo de software é usado para facilitar o processo tedioso de conversão e optimização de código (Aho et al., 2006). Um compilador costuma aceitar e/ou devolver vários tipos de objetos como código-fonte, *bytecode*, código de máquina, *description objects*, etc. Existem também vários tipos de compiladores: *Native compilers*, *bytecode compilers*, Just in Time (JIT) *compilers*, *Source-to-source compilers* e *hardware compilers*.

2.2 TIPOS DE CÓDIGO

2.2.1 *Código-fonte*

Um dos objetos de entrada mais comuns usados por compiladores é o código-fonte. O código-fonte é um conjunto de símbolos legíveis com uma certa ordem que representam instruções numa certa linguagem de programação (T. L. I. Project, 2006).

```d
import std.stdio;

void main()
{
    writeln("Olá mundo!");
}
```

Listagem 1: Exemplo de código-fonte escrito na linguagem de programação D

Na Listagem 1 pode-se ver um exemplo de código-fonte escrito na linguagem de programação D aceite pela sua implementação de compilador de referência Digital Mars D (DMD).

2.2.2 *Bytecode*

Bytecode é uma representação de instruções desenhado para ser interpretado por uma máquina virtual. Estas instruções costumam ser de baixo-nível e eficientes de forma a que sejam interpretadas por software (Christensson, 2018). Normalmente este tipo de instruções é uma forma de mapear instruções genéricas virtuais em instruções especificas de hardware.

Por exemplo, a linguagem de programação Java está desenhada para correr numa máquina virtual chamada Java Virtual Machine (JVM).

```java
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Listagem 2: Exemplo de código-fonte em Java

Na Listagem 2 podemos ver um exemplo de código escrito na linguagem de programação Java que, quando compilado, gera bytecode para a máquina virtual JVM.

```
stack=1, locals=2, args_size=1
    start local 0
     0: aload_0
     1: invokespecial #1
     4: iconst_0
     5: istore_1
    start local 1
     6: iinc          1, 1
     9: return
    end local 1
    end local 0
```

Listagem 3: Exemplo de bytecode Java

Na Listagem 3 pode-se ver um exemplo legível de bytecode Java que corre na máquina virtual JVM. Esta representação, gerada com o recurso à ferramenta `javap`[1], é semanticamente equivalente ao código-fonte apresentado na Listagem 2.

2.2.3 *Código de máquina*

O código de máquina é uma representação da linguagem interpretada por uma determinada arquitetura de um Central Processing Unit (CPU). Esta é representada por várias instruções simples que são interpretadas e executadas pelo CPU. Este tipo de código é um dos objetos de saida mais conhecidos de um compilador (*What is Machine Language? | Webopedia* 2021).

```
section .text
    global  _start
_start:
    mov     edx,len
    mov     ecx,msg
    mov     ebx,1
    mov     eax,4
    int     0x80
section .data
msg db      "Hello, world!",0xa
len equ     $ - msg
```

Listagem 4: Exemplo código de máquina x86 em NASM

Na Listagem 4 é possivel ver um exemplo de código de máquina legível da arquitetura x86 na linguagem de *assambly* do *assembler* Netwide Assembler (NASM).

[1] https://javap.yawk.at/

2.3 TIPOS DE COMPILADORES

2.3.1 *Native compilers*

Um compilador nativo é um compilador Ahead of Time (AOT) que transforma um determinado código-fonte em código de máquina especifico para uma determinada arquitetura de CPU. O objetivo principal destes compiladores é ter a possibilidade de correr código relativamente de alto nivel de forma nativa em vez de este mesmo código ser interpretado por *software*. As vezes é necessário desenhar sistemas de alta performance e com o recurso a este tipo de compiladores, a tarefa torna-se muito mais simplificada.

Exemplos

Um exemplo notável de um compilador nativo é o famoso GNU Compiler Collection (GCC). Este compilador conta com um vasto número de implementações para varias linguagens de programação incluindo Ada, C, C++, Fortran, Go, D, etc. Este compilador é muito usado principalmente pela sua altissima performance e implementação do seu *backend* para multiplas arquiteturas Complex Instruction Set Computer (CISC) e Reduced Instruction Set Computer (RISC) como x86, Acorn RISC Machine (ARM), Alf and Vegard's RISC (AVR), Microprocessor without Interlocked Pipeline Stages (MIPS), etc (F. S. Foundation, 2021).

Outro exemplo notável é o compilador Clang desenvolvido pela equipa do projeto Low Level Virtual Machine (LLVM). Este é uma implementação mais recente do *frontend* de C, C++, Objective-C e Objective-C++ e teve como objetivo substituir o conjunto de compiladores GCC. (L. Project, 2021) Uma das grandes vantagens em relação ao GCC é o facto de o LLVM ter uma representação intermédia *standard*, permitindo modelarizar qualquer *frontend* de uma linguagem usando este *backend*.

2.3.2 *Bytecode compilers*

Um compilador de *bytecode* é um compilador muito semelhante aos compiladores nativos mas em vez de gerarem código de máquina, geram uma representação intermédia em *bytecode* que normalmente não é diretamente executável.

Exemplos

Um exemplo muito conhecido é o compilador da linguagem de programação *Java*, `javac`. Este compilador compila código-fonte *Java* para *Java bytecode* com o intuito de este correr na máquina virtual JVM (Lindholm et al., 2015).

Outro exemplo notável, embora não muito conhecido como sendo um compilador, é o interpretador de referencia da linguagem Python, o CPython[2]. Este interpretador contem um compilador que transforma código-fonte escrito em Python em uma linguagem de *bytecode* que por sua vez é imediatamente interpretado (P. S. Foundation, 2021).

2.3.3 *Just-in-time compilers*

Um Just in Time (JIT) *compiler* é um compilador que envolve compilar, em runtime, pedaços ou mesmo todo o código para código de máquina. Este código pode ser em forma de código-fonte ou, o mais comum, em *bytecode*. O principal objetivo deste tipo de compiladores é optimizar rotinas frequentemente usadas e assim ser possivel de executa-las diretamente na maquina em vez de, por exemplo, numa máquina virtual, removendo o *overhead* de interpretar o *bytecode* por *software* (Aycock, 2003).

Exemplos

Mais uma vez, Java é um exemplo de uma linguagem de programação que tira partido de JIT *compilers* para optimizar o seu *bytecode* na sua implementação de máquina virtual chamada Java HotSpot (OpenJDK, 2021).

2.3.4 *Source-to-source compilers*

Um *source-to-source compiler*, tambem conhecido como transpilador é um compilador que transforma código-fonte em outro código-fonte. No passado, estes transpiladores foram usados muito para manter compatibilidade de código enquando uma parte era migrada para uma tecnologia e linguagens mais recentes. Hoje em dia, estas ferramentas são usadas para converter código-fonte de uma linguagem mais *type-safe* e *memory-safe* para linguagens mais dinamicas ou adicionar suporte

[2] https://github.com/python/cpython

para funcionalidades como Object Oriented Programming (OOP) (Gurumoorthy P, 2019).

Exemplos

Um exemplo muito recente e conhecido é o transpilador de TypeScript. Este transpilador converte código-fonte TypeScript que é strongly-typed com a mesma semântica de JavaScript em código-fonte JavaScript, poorly-typed. O objetivo é gerar código JavaScript com o type-checking adicional que o JavaScript não tem (Microsoft, 2021).

```
function foo(obj: Object) {}

foo({}) // OK
foo(1) // Error
```

Listagem 5: Exemplo código em TypeScript

```
function foo(obj) {}

foo({}) // OK
foo(1) // OK
```

Listagem 6: Exemplo código em JavaScript

Na Listagem 5 e 6 pode-se observar dois exemplos de código-fonte semânticamente equivalentes de acordo com as regras de JavaScript, embora, na Listagem 5 a ultima linha não ira compilar num compilador de TypeScript, uma vez que o tipo do valor do argumento passado para a função não correspondem.

2.3.5 *Hardware compilers*

Um *hardware compiler* é uma ferramenta que compila código-fonte em *description objects* cujo objetivo é descrever as funcionalidades lógicas em portas lógicas fisicas. Este tipo de compiladores é muito usado na programação de Field Programmable Gate Array (FPGA)s e Complex Programmable Logic Device (CPLD)s ou na construção de Application-specific Integrated Circuit (ASIC)s.

Exemplos

Muitos dos exemplos de *hardware compiler* são implementações especificas das marcas produtoras de *hardware* com o recurso a Hardware Description Language (HDL) como Verilog e VHSIC Hardware Description Language (VHDL).

Apesar de maior parte das implementações serem especificas, existem implementações genéricas como simuladores de hardware com a linguagem VHDL. Um dos exemplos é o simulador G Hardware Design Language (GHDL) que compila e executa código VHDL num computador convencional, simulando o hardware (Gingold, 2017).

3

FASES DE COMPILAÇÃO

Porque uma ferramenta como um compilador tem uma complexidade enorme, este normalmente é desenhado com várias fases de compilação. Porem existem compiladores com apenas uma fase de compilação, chamados *one-pass compilers*, que dependem da complexidade da linguagem. Antigamente esta era uma abordagem muito usada pois os recursos da maquina eram muito limitativos. Por exemplo, os compiladores de Pascal eram normalmente desenhados apenas com uma fase de compilação, pois a linguagem foi especificamente desenhada para compilar numa só fase dai a ordem das declarações ser obrigatória (Keleshev, 2020).

Com o passar dos anos os compiladores foram evoluindo e hoje em dia são divididos em duas grandes partes no *frontend* e no *backend*. O *frontend* contem principalmente o *lexer*, o *parser* e processa a semântica da linguagem. O *backend* contem principalmente o *optimizer* e o *target code generation*.

A grande razão para este facto é o aumento do número de arquiteturas que um compilador necessita de suportar hoje em dia. Separando o *frontend* do *backend*, os compiladores conseguem-se focar na implementação especifica das regras de uma determinada linguagem no *frontend* e focar-se na optimização e geração de código especifico para cada ambiente usando uma representação genérica. Um dos grandes exemplos de compiladores que implementão esta estrutura são o GCC e o Clang do projeto LLVM. Estas duas implementações são revolucionarias, principalmente o LLVM que implementa um *backend* genérico e linguagens novas como Rust[1] e Zig[2] apenas implementam o *frontend*.

[1] https://www.rust-lang.org/
[2] https://ziglang.org/

3.1 FRONTEND

3.1.1 *Pré processador*

O pré processador é uma fase de compilação implementada pelos compiladores para linguagem com suporte para *macros*, embora muitas implementações não necessitem de implementar um pré processador. Esta fase é útil para interpretar e pré processar diretivas especificas de pré processamento, como por exemplo o `#include`, `#define` e `#pragma` da linguagem de programação C. (GNU, 2021)

```c
/* ficheiro: foo.h */
#if defined(__LP64__) || defined(_LP64)
    #define IS_MARCH_64   1
#else
    #define IS_MARCH_64   0
#endif

/* ficheiro: foo.c */
#include "foo.h"

int runtime_var = IS_MARCH_64;
```

Listagem 7: Exemplo de ficheiros com diretivas

Na Listagem 7 pode-se observar dois ficheiros com diretivas de pré processamento. Neste exemplo, se o ficheiro `foo.c` for compilado, esta fase trata de copiar o conteudo do ficheiro `foo.h` e colar no inicio do ficheiro `foo.c` com o recurso à diretiva `#include`, verifica a existencia da definição de `__LP64__` e `_LP64` com o recurso à diretiva `#if` e por fim copia o valor de `IS_MARCH_64` e cola sempre que o token `IS_MARCH_64` é usado.

```c
/* ficheiro: foo.c */
int runtime_var = 0;
```

Listagem 8: Exemplo de ficheiro em 32 bits

```c
/* ficheiro: foo.c */
int runtime_var = 1;
```

Listagem 9: Exemplo de ficheiro em 64 bits

Na listagem 8 e 9 pode-se observar o resultado do ficheiro `foo.c` depois de passar a fase de pré processamento, dependendo da arquitetura *target*.

3.1.2 *Análise lexical*

A analise lexical é uma fase praticamente usada em todos os compiladores e consiste no processo simples de separar o *input* em vários tokens. A análise lexical é realizada com o recurso a um *lexer* e pode ser constituido por um *scanner* e um *tokenizer*. O *scanner* trata de converter o input em varios *lexemes* (conceito de sequencia de caracteres em ciencias da linguagem). O *tokenizer* trata de converter os *lexemes* em *tokens* (Trim, 2009). Em compiladores mais recentes, este processo é feito numa só vez e pode não existir o conceito de *scanner*.

```
1 if (foo == "foo") /* when true */
2 {
3     foobar = 1 + 2;
4     break;
5 }
```

Listagem 10: Exemplo de uma condição if em C

Na Listagem 10 pode-se observar um exemplo de código com uma condição if, um comentario e logica para se essa condição for verdadeira.

Tipo de *token*	**Valores**
delimitador	`{`, `}`, `(`, `)`, `;`
operador	`==`, `=`, `+`
identificador	`foo`, `foobar`
keyword	`if`, `break`
literal	`"foo"`, `1`, `2`
comentarios	`/* when true */`

Tabela 1: Tabela com os *tokens* da Listagem 10

Na Tabela 1 pode-se obeservar os *tokens* processador pelo *tokenizer* de acordo com a Listagem 10.

3.1.3 *Análise sintática*

A análise sintática é uma fase do compilador que pega nos *tokens* previamente gerados e analisa sintáticamente se a organização dos *tokens* está conforme uma determinada gramática e assim gera uma Abstract Syntax Tree (AST). Para isso, o

compilador usa um *parser* para realizar essa tarefa. Existe vários tipos de *parsers* para o efeito, *top-down parsers* (LL, *Left-to-right Leftmost derivation*) e *bottom-up parsers* (LR, *Left-to-right, Rightmost derivation in reverse*) (Grune e Jacobs, 2008). Alguns destes tipos de *parsers* têm vantagens e desvantagens em relação a performance e capacidade de interpretar determinadas gramáticas, mas não é o foco deste documento estudar cada tipo de *parser*.

```
<IfStatement> ::=
    "if" "(" <IfCondition> ")" <ThenStatement>
    | "if" "(" <IfCondition> ")" <ThenStatement> "else" <ElseStatement>
```

Listagem 11: Gramática do statement if da linguagem de programação D

Na Listagem 11 pode-se observar a gramática simplificada em Backus-Naur Form (BNF) referente ao statement if da linguagem de programação D[3].

A partir da gramática definida na Listagem 11 é possivel verificar se um conjunto de *tokens* cumpre a gramatica e posteriormente construir uma árvore de sintaxe referente ao *if statement* com os valores contidos em cada nó.

```
if(foo)
    foo = 1;
else
    bar = 1;
```

Listagem 12: Exemplo de código em linguagem D que cumpre com a sua gramática

Por exemplo, na Listagem 12 pode-se observar código-fonte que genéricamente cumpre com a gramática definida.

```
if)foo(
    foo = 1;
else
    bar = 1;
```

Listagem 13: Exemplo de código em linguagem D que **não** cumpre com a sua gramática

Ja na Listagem 13 pode-se observar que o código-fonte não cumpre com a gramática definida uma vez que claramente os *tokens* delimitadores estão trocados.

[3] https://dlang.org/spec/grammar.html#IfStatement

Depois de verificar a conformidade com a gramática é gerada uma AST com a estrutura da gramática e os valores associados aos tokens.

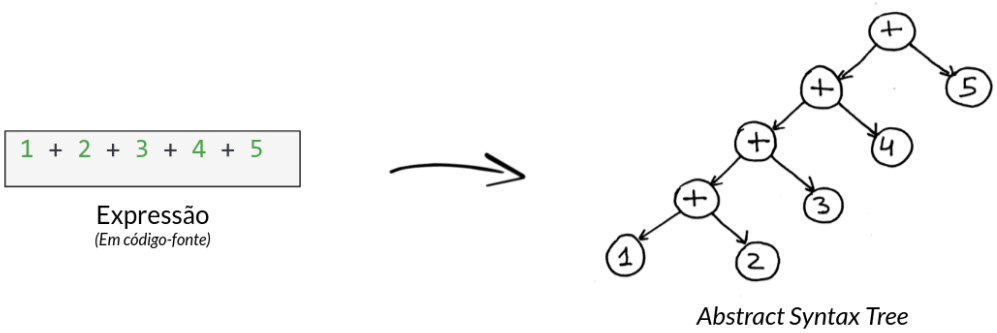

Figure 1: Comparação entre o código-fonte e a AST gerada (Spivak, 2015)

Na Figura 1 pode-se observar uma comparação do código-fonte e a AST gerada pelo *parser*.

3.1.4 *Análise Semântica*

A análise semântica é a fase mais complexa do *frontend* de um compilador. Após gerar a árvore de sintaxe a fase da semântica é responsavel por resolver o sistema de tipos, isto é inferir os tipos omitidos, bem como *forward references* para tipos apenas conhecidos em outros nós. Nesta fase o compilador gera novas árvores de sintaxe mais refinadas. Devido à complexidade desta fase, existem muitos compiladores que subdividem esta fase em várias fases de semântica, onde cada uma gera uma nova árvore de sintaxe(Spivak, 2017).

Sem entrar em muito detalhe, nesta fase para resolver o sistema de tipos é comum usar uma *symbol lockup table*. Esta tabela vai sendo preenchida consoante necessário com a referencia do símbolo, o seu tipo, a sua localização e o seu valor. Muitas implementações têm mais ou menos informação na tabela de símbolos dependendo da sua necessidade, mas genéricamente esta é uma informação necessária.

```
auto foo(int bar)
{
    auto foobar = 5;
    return foobar + bar;
}
```

Listagem 14: Exemplo de código em D antes da semântica

Por exemplo, na Listagem 14 pode-se observar um exemplo de código-fonte antes da fase da semântica. Particularmente, pode-se verificar que o tipo de retorno e o tipo das variaveis locais são desconhecidos, com o recurso à *keyword* `auto`.

Nome	**Tipo**	**Scope**
`foo`	função, inteiro	global
`bar`	inteiro	parametro
`foobar`	inteiro	local

Tabela 2: *Symbol Lookup Table* da Listagem 14

Durante a fase da semântica, a Tabela 2 que representa a tabela de símbolos é construida. Com esta informação é assim possivel "substituir"a *keyword* `auto` pelo tipo correspondente.

```
1 import object;
2 pure nothrow @nogc @safe int foo(int bar)
3 {
4   int foobar = 5;
5   return foobar + bar;
6 }
```

Listagem 15: Exemplo de código em D depois da semântica

Na Listagem 15 pode-se observar uma representação de código em D depois da fase da semântica. Verifica-se que o tipo de retorno e o tipo das variáveis locais foi realmente inferido. Outra informação adicional foi inferida, como por exemplo, o compilador anotou que a função é pura, não lança nenhuma excepção, não usa o Garbage Collector (GC) e é considerada *memory-safe*.

3.1.5 *Intermediate Code Generation*

Após gerar a árvore de sintaxe final existe uma fase que gera o Intermediate Representation (IR) chamada *intermediate code generation*. O IR é apenas uma representação intermédia em forma de *bytecode* que tem o mesmo significado semântico do código representado pela árvore de sintaxe final. Esta fase é essencial para fazer a ponte entre o *frontend* e o *backend* do compilador. Este código intermédio é em forma de operações muito simplificadas, normalmente de baixo nível e é independente do ambiente da máquina (Walker, 2003).

```
int foo(int bar)
{
  return bar + 5;
}
```

Listagem 16: Exemplo de código em D

```
%bar = alloca i32, align 4
store i32 %bar_arg, i32* %bar, align 4
%1 = load i32, i32* %bar, align 4
%2 = add i32 %1, 5
ret i32 %2
```

Listagem 17: Exemplo de IR representativo da Listagem 16 em LLVM

Na Listagem 16 e 17 pode-se observar a equivalência entre o código-fonte da Listagem 16 e o IR respetivo em LLVM[4] na Listagem 17.

3.2 BACKEND

3.2.1 *Optimizer*

O *optimizer* é uma fase opcional do *backend* de um compilador onde este pode optimizar certas rotinas quando ao espaço em disco, memória usada e performance em termos de número de instruções executadas. Esta fase pode incluir vários tipos de optimizações onde algumas delas é o utilizador que escolhe usa-las ou não. Optimizações como remoção de variáveis locais nunca usadas, remoção de bolhas no *pipelining* do CPU *target*, evitar o uso de instruções *jump*, *loop unrolling*, *function inlining*, etc.

```
%bar = alloca i16, align 2
store i16 %bar_arg, i16* %bar, align 2
%1 = load i16, i16* %bar, align 2
%negval = sub i16 0, %1
ret i16 %negval
```

Listagem 18: Exemplo de IR **não** optimizado

[4] https://llvm.org/docs/LangRef.html

```
%negval = sub i16 0, %bar_arg
ret i16 %negval
```

Listagem 19: Exemplo de IR optimizado

Na Listagem 18 e 19 pode-se observar a diferença gerada entre versões do IR não optimizado e optimizado, respetivamente. Na Listagem 19 verifica-se que o *optimizer* decidiu não alocar uma cópia da variavel `bar_arg` na *stack*, uma vez que essa cópia nunca é usada, e assim nunca modificada, é então preferível usar a sua referência diretamente.

3.2.2 *Target Code Generation*

O *Target Code Generation* é normalmente a ultima fase de compilação onde o compilador transforma o IR em código de máquina, específico da arquitetura *target*.

```
movl  %edi, %eax
negl  %eax
retq
```

Listagem 20: Exemplo de *Assembly* x86_64

Na Listagem 20 pode-se observar a representação em *assembly* x86_64 da Listagem anterior 19.

4

CONCLUSÃO

Os compiladores estão cada vez a ficar mais complexos e é necessário entender como esta ferramenta funciona internamente para melhor usa-la. Compreender compiladores não se trata só de curiosidade mas também entender a practicalidade do tipo de compilador e linguagem de programação que se escolhe para desenvolver um projeto.

4.1 RECOMENDAÇÕES

Apesar de este documento falar de uma forma breve de como funciona um compilador, não é aprofundado análises de árvores de sintaxe. Este é um assunto importante para entender outras funcionalidades de um compilador, como por exemplo, algoritmos de optimização com Deterministic Finite Automaton (DFA)s, aplicabilidade de *turing machines* para resolver certos problemas semânticos, etc. A abordagem deste tópico é um pouco mais extensiva e implica certos pré-requisitos teóricos e matemáticos como teoria da computação e teoria de grafos.

Caso o leitor queira explorar este assunto mais extensivamente, recomenda-se a consulta do livro Aho et al. (2006) e do livro Parsons (2003).

REFERÊNCIAS BIBLIOGRÁFICAS

Aho, Alfred et al. (ago. de 2006). *Compilers: Principles, Techniques, and Tools*. xn–3ug : xn–2ug Addison Wesley. URL: https://www.amazon.com/Compilers-Principles-Techniques-Alfred-Aho/dp/0321493028.

Aycock, John (jun. de 2003). «A Brief History of Just-In-Time». Em: *ACM Comput. Surv* 35.2. [Online; acedido a 5. Jul. 2021], pp. 97–113. DOI: 10.1145/857076.857077. URL: https://prism.ucalgary.ca/handle/1880/45368.

Christensson, P. (jan. de 2018). *Bytecode Definition*. [Online; acedido a 4. Jul. 2021]. URL: https://techterms.com/definition/bytecode.

Foundation, Free Software (jun. de 2021). *GCC, the GNU Compiler Collection - GNU Project - Free Software Foundation (FSF)*. [Online; acedido a 4. Jul. 2021]. URL: https://gcc.gnu.org.

Foundation, Python Software (jul. de 2021). *Welcome to Python.org*. [Online; acedido a 5. Jul. 2021]. URL: https://www.python.org/about.

Gingold, Tristan (dez. de 2017). *GHDL Main/Home Page*. [Online; acedido a 5. Jul. 2021]. URL: http://ghdl.free.fr.

GNU (jun. de 2021). *Overview (The C Preprocessor)*. [Online; acedido a 5. Jul. 2021]. URL: https://gcc.gnu.org/onlinedocs/cpp/Overview.html.

Grune, Dick e Ceriel J. H. Jacobs (2008). *Parsing Techniques*. New York, NY, USA: Springer-Verlag. ISBN: 978-0-387-20248-8. DOI: 10.1007/978-0-387-68954-8.

Gurumoorthy P, Arvind Padmanabhan (jan. de 2019). «Transpiler». Em: *Devopedia*. [Online; acedido a 5. Jul. 2021]. URL: https://devopedia.org/transpiler.

Keleshev, Vladimir (Mai de 2020). *One-pass Compiler Primer*. [Online; acedido a 5. Jul. 2021]. URL: https://keleshev.com/one-pass-compiler-primer.

Lindholm, Tim et al. (Fev de 2015). *The Java® Virtual Machine Specification*. [Online; acedido a 5. Jul. 2021]. URL: https://docs.oracle.com/javase/specs/jvms/se8/html.

Microsoft (jul. de 2021). *Typed JavaScript at Any Scale*. [Online; acedido a 5. Jul. 2021]. URL: https://www.typescriptlang.org.

OpenJDK (jun. de 2021). *HotSpot Group*. [Online; acedido a 5. Jul. 2021]. URL: https://openjdk.java.net/groups/hotspot.

Parsons, Thomas W (2003). *Introduction to compiler construction*. Computer Science Press. ISBN: 0-7167-8261-8. URL: https://www.amazon.com/Introduction-Compiler-Construction-Thomas-Parsons/dp/0716782618.

Project, LLVM (jul. de 2021). *The LLVM Compiler Infrastructure Project*. [Online; acedido a 4. Jul. 2021]. URL: https://llvm.org.

Project, The Linux Information (Fev de 2006). *Source code definition by The Linux Information Project*. [Online; acedido a 4. Jul. 2021]. URL: http://www.linfo.org/source_code.html.

Spivak, Ruslan (dez. de 2015). «Let's Build A Simple Interpreter. Part 7: Abstract Syntax Trees». Em: *Ruslan's Blog*. [Online; acedido a 6. Jul. 2021]. URL: https://ruslanspivak.com/lsbasi-part7.

— (abr. de 2017). «Let's Build A Simple Interpreter. Part 13: Semantic Analysis.» Em: *Ruslan's Blog*. [Online; acedido a 6. Jul. 2021]. URL: https://ruslanspivak.com/lsbasi-part13.

Trim, Craig (out. de 2009). *The Art of Tokenization (Language Processing)*. [Online; acedido a 5. Jul. 2021]. URL: https://web.archive.org/web/20190530155339/https://www.ibm.com/developerworks/community/blogs/nlp/entry/tokenization?lang=en.

Walker, David (abr. de 2003). *CS320: Compilers: Intermediate Representation*. [Online; acedido a 6. Jul. 2021]. URL: https://www.cs.princeton.edu/courses/archive/spr03/cs320/notes/IR-trans1.pdf.

What is Machine Language? | Webopedia (jun. de 2021). [Online; acedido a 4. Jul. 2021]. URL: https://www.webopedia.com/definitions/machine-language.