SOFTWARE ENGINEERING

Compiler vs. Interpreter: Know The Difference And When To Use Each Of Them

Types and use cases of compilers and interpreters

Rakia Ben Sassi · Follow
Published in Better Programming · 7 min read · Jan 19, 2021

350

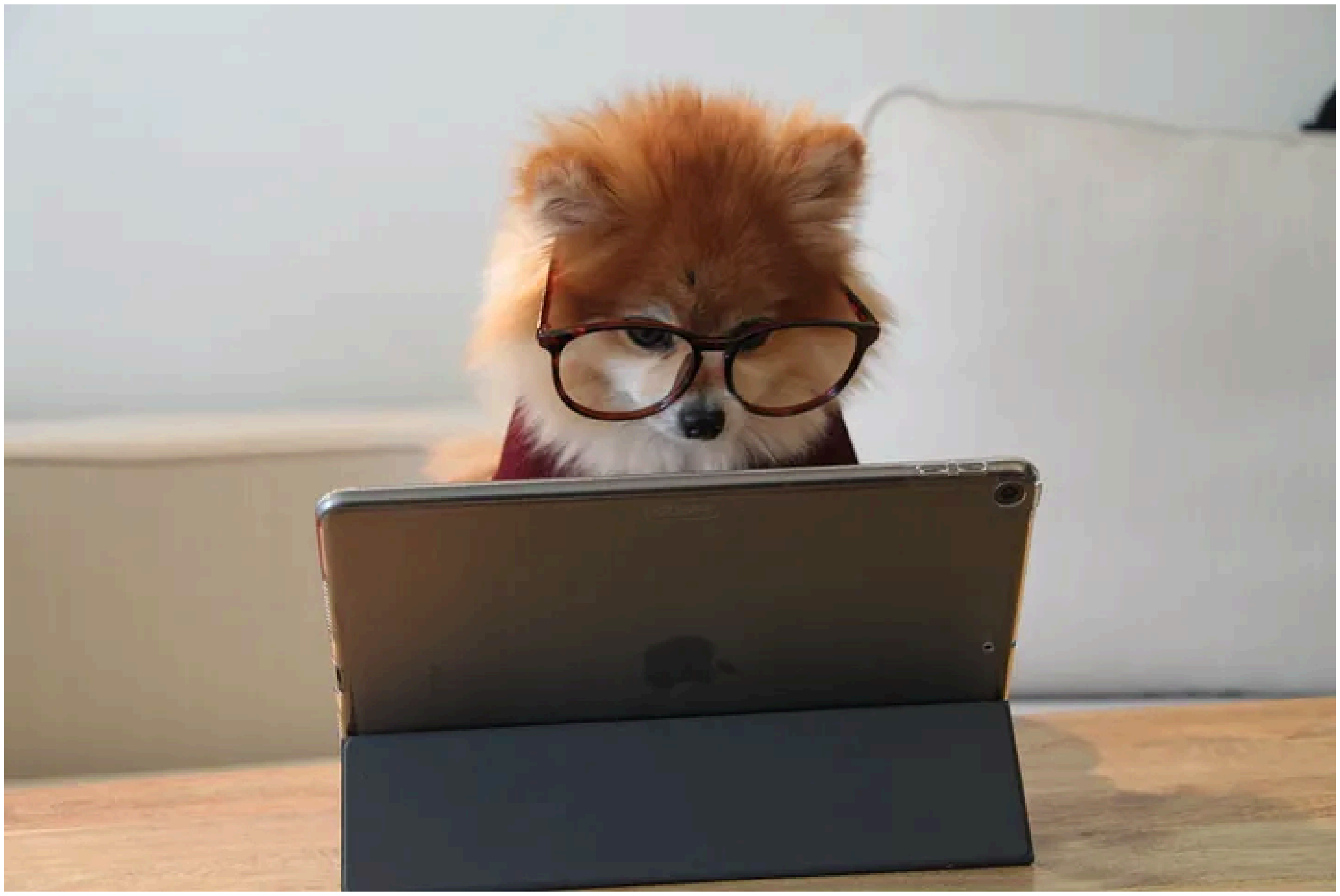


Photo by Cookie the Pom on Unsplash

I still remember a discussion with a colleague of mine in which I said, “That’s the transpiler,” and he replied: “Trans…what?”

If you have never heard that name, you’re not alone. As developers, we all get used to writing code in a high-level language that humans can understand. However, computers can only understand a program written in a binary system known as machine code.

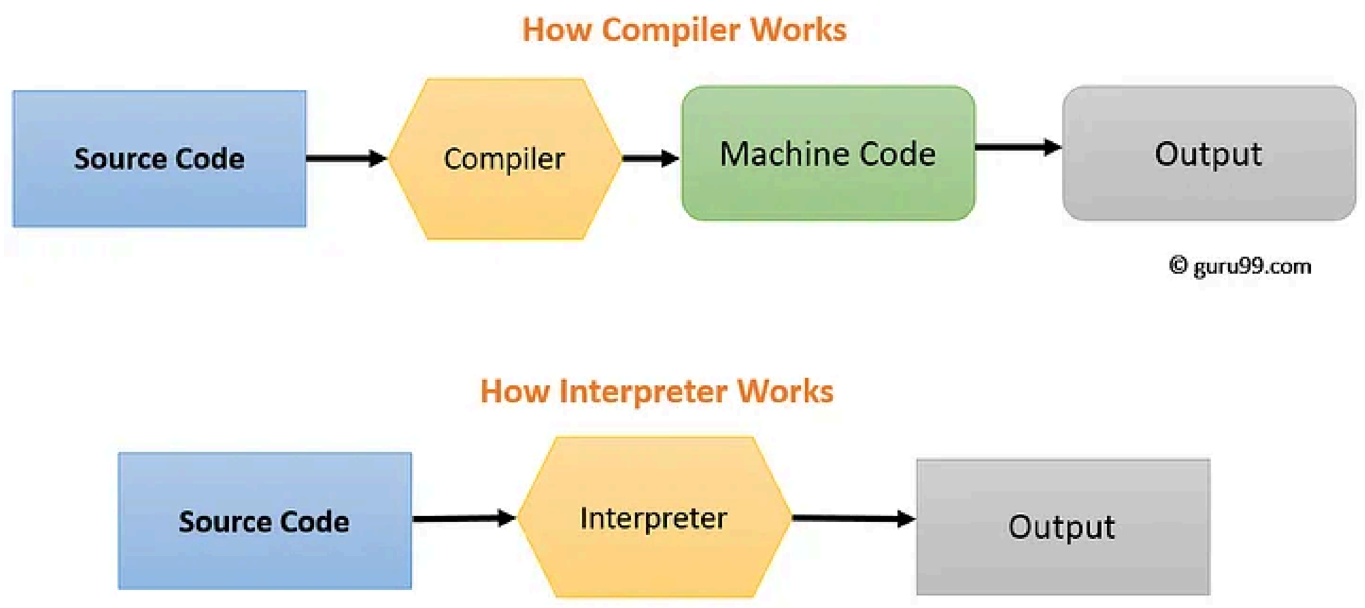
To speak to a computer in its non-human language, we came up with two solutions: interpreters and compilers. Ironically, most of us know very little about them, although they belong to our daily coding life.

In this post, I’ll dive into the journey of translating a high-level language into a machine code ready for execution. I’ll focus on the inner working of the two key players in this game — the compiler and the interpreter — and break down the related concepts.

The Journey From High- to Lower-Level Language

Compilers and interpreters have long been used as computer programs to transform code. But they work in different ways:

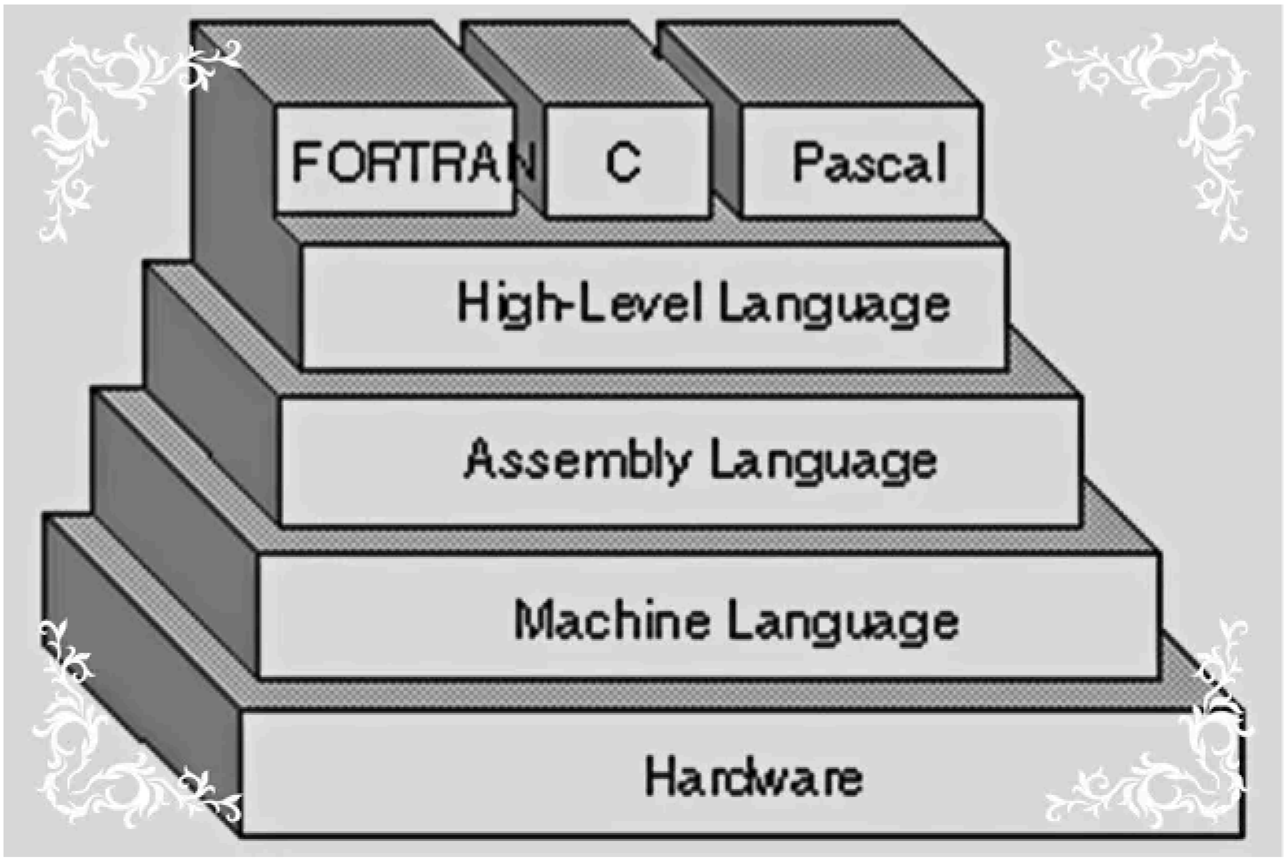
- A compiler translates a code written in a high-level programming language into *a lower-level language like assembly language, object code, and* machine code (binary 1 and 0 bits). It converts the code ahead of time before the program runs.
- An interpreter translates the code line by line when the program is running. You've likely used interpreters unknowingly at some point in your work career.



Compiler vs Interpreter

Both compilers and interpreters have pros and cons:

- A compiler takes an entire program and a lot of time to analyze the source code, whereas the interpreter takes a single line of code and very little time to analyze it.
- A compiled code runs faster while interpreted code runs slower.
- A compiler displays all errors after compilation. If your code has mistakes, it will not compile. But the interpreter displays errors of each line one by one.
- Interpretation does not replace compilation completely.
- Compilers can contain interpreters for optimization reasons like faster performance and smaller memory footprint.



Assembly Language

A high-level programming language is usually referred to as "compiled language" or "interpreted language." However, in practice, they can have both compiled and interpreted implementations. C, for example, is called a compiled language, despite the existence of C interpreters. The first JavaScript engines were simple interpreters, but all modern engines use just-in-time (JIT) compilation for performance reasons.

Types of Interpreter

Interpreters were used as early as 1952 to ease programming and also used to translate between low-level machine languages. The first interpreted high-level language was Lisp. Python, Ruby, Perl, and PHP are other examples of programming languages that use interpreters.

Below is a non-exclusive list of interpreter's types:

1. Bytecode interpreter

The trend toward bytecode interpretation and just-in-time compilation blurs the distinction between compilers and interpreters.

> *"In a bytecode interpreter each instruction starts with a byte, and therefore bytecode interpreters have up to 256 instructions, although not all may be used. Some bytecodes may take multiple bytes, and may be arbitrarily complicated."*
>
> *— Wikipedia*

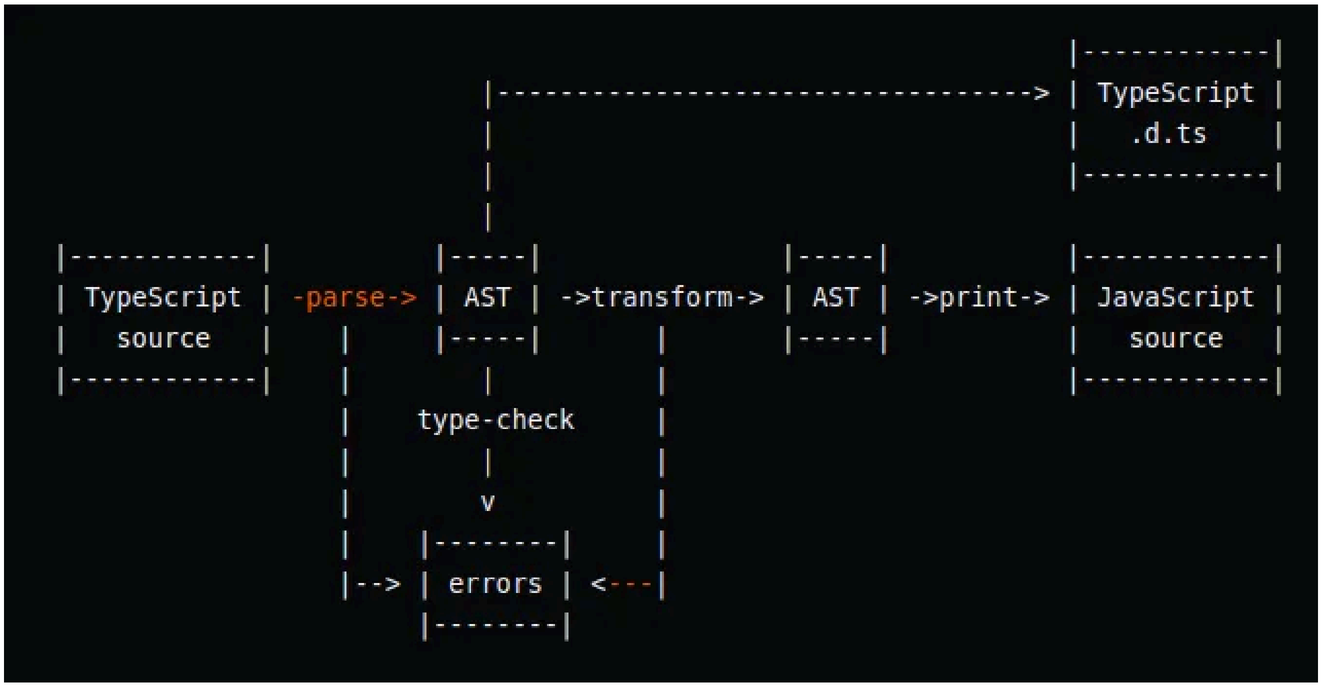
2. Threaded code interpreter

Unlike bytecode interpreters, threaded code interpreters use pointers instead of bytes. Each instruction is a word pointing to a function or an instruction sequence, possibly followed by a parameter. The number of different instructions is limited by available memory and address space.

The Forth code — used in Open Firmware systems — is a classical example of threaded code. The source code is compiled into a bytecode known as “F code”, then interpreted by a virtual machine.

3. Abstract syntax tree interpreter

If you’re a TypeScript developer and you have some insights about the TypeScript architecture, you may have heard the acronym AST for Abstract Syntax Tree.



TypeScript Transpiler Architecture

AST is an approach to transform the source code into an optimized abstract

AST keeps the global program structure and relations between statements. This allows the system to perform better analysis during runtime and makes AST a better intermediate format for just-in-time compilers than bytecode representation.

However, for interpreters, AST causes more overhead. Interpreters walking the abstract syntax tree are slower than those generating bytecode.

4. Just-in-time compilation

Just-in-time compilation (JIT) is a technique in which the intermediate representation is compiled to native machine code at runtime.

Types of Compiler

1. Cross-compiler

A compiler running on a computer whose CPU or operating system differs from the one on which the code it produces will run.

2. Native compiler

A compiler producing an output that would run on the same type of computer and operating system as the compiler itself.

3. Bootstrap compiler

A compiler written in the language that it intends to compile.

4. Decompiler

A decompiler translates code from a low-level language to a higher level one.

5. Source-to-source compiler (Transpiler)

It's a program that translates between high-level languages. This type of compilers is also known as a transcompiler or transpiler.

Examples:

- Emscripten: transpiles C/C++ to JavaScript.
- Babel: transpiles JavaScript code from ES6+ to ES5.
- Cfront: the original compiler for C++ (from around 1983). It used C as its target language and created C code with no indent style and no pretty C intermediate code, since the generated code was usually not intended to be readable by humans.

6. A language rewriter

This is usually a program translating form of expressions without a change of language.

7. Bytecode compiler

A compiler that translates a high-level language into an intermediate simple language that can be interpreted by a bytecode interpreter or a virtual machine.

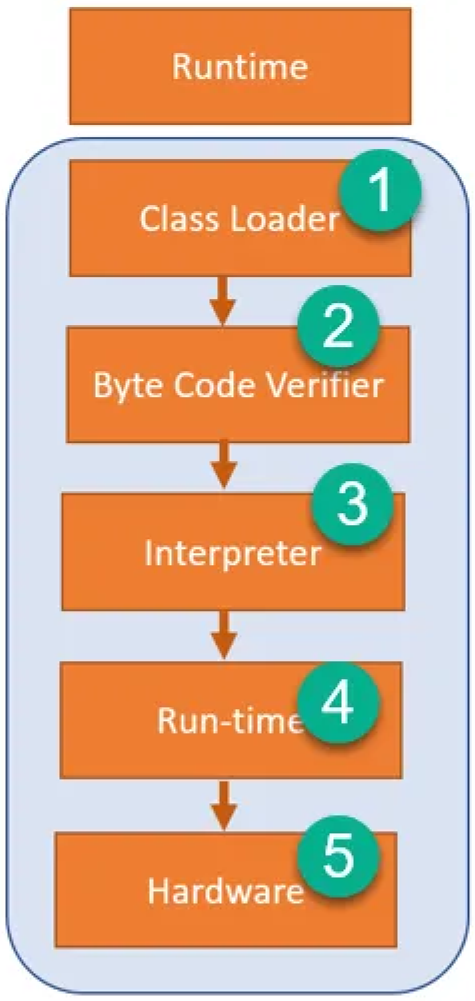
Examples: Bytecode compilers for Java and Python.

8. Just-in-time compiler (JIT Compiler)

JIT compiler defers compilation until runtime. It generally runs inside an interpreter.

Examples:

- The earliest published JIT compiler is attributed to *LISP* in 1960.
- The latter technique appeared in languages such as Smalltalk in the 1980s.
- Then JIT compilation has gained mainstream attention amongst modern language like Java, .NET Framework, Python, and most modern JavaScript implementations.



JRE (Java Runtime Environment) Functionality

In Java, source files are first compiled and converted into `.class` files which contain Java bytecode (highly optimized set of instructions), then a bytecode interpreter executes the bytecode, and later the JIT compiler translates the bytecode to machine code.

Java bytecode can either be interpreted at runtime by a virtual machine, or compiled at load time or runtime into native code. Modern JVM implementations use the compilation approach, so after the initial startup time the performance is equivalent to native code.

9. AOT Compilation

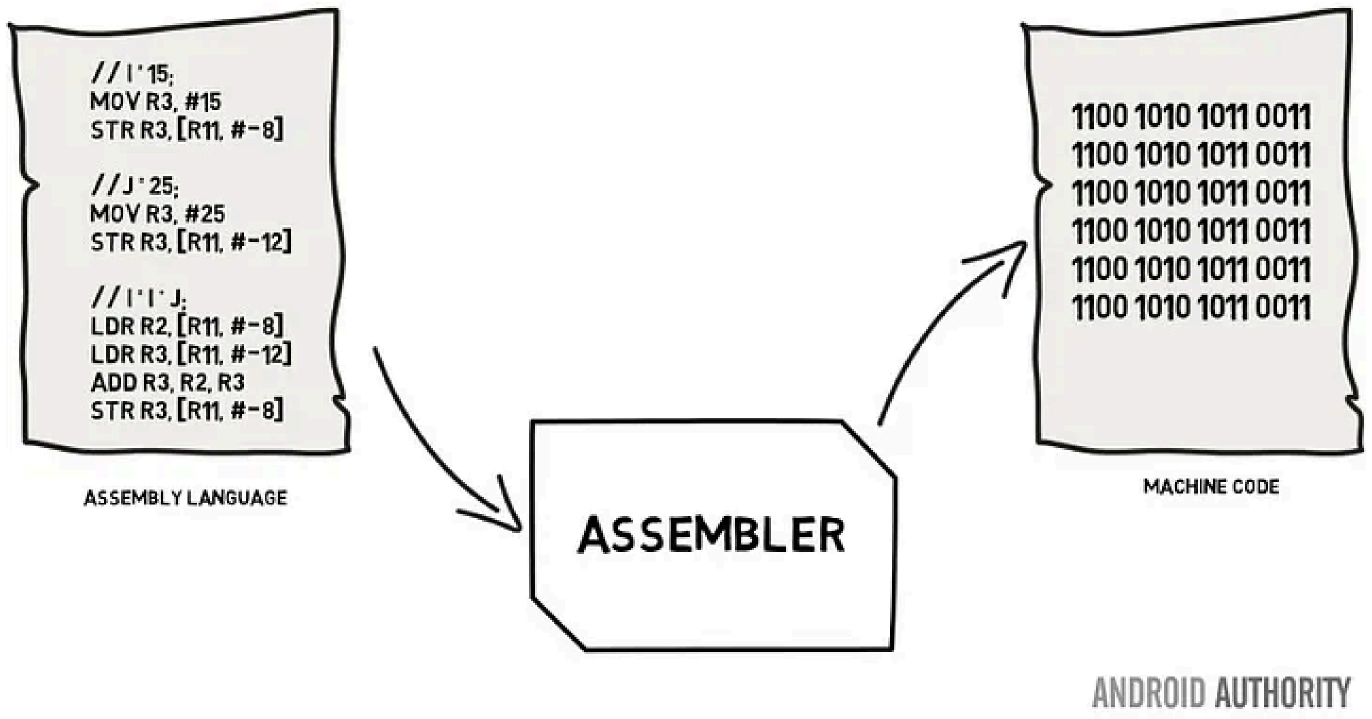
Ahead-of-time (AOT) compilation is the approach of compiling a higher-level programming language, or an intermediate representation such as Java bytecode, before the runtime.

Example:

The Angular framework uses an ahead-of-time (AOT) compiler to transform HTML and TypeScript code into JavaScript code during the build time to provide a faster rendering later on the browser when the code is running.

10. Assembler

An assembler translates human-readable assembly language into machine code. This compilation process is called assembly. The inverse program that converts machine code to assembly language is called a disassembler.



Assembler

An assembly language (abbreviated ASM) is a low-level programming language in which there is a dependence on the machine code instructions. That's why every assembly language is designed for exactly one specific computer architecture.

Takeaway

Both compilers and interpreters are computer programs that convert a code written in a high-level language into a lower-level or machine code understood by computers. However, there are differences in how they work and when to use them.

Even if you're not going to implement the next compiler or interpreter, these insights should help to improve your knowledge of the tools you use as a developer every day.

🧠💡 I write about engineering, technology, and leadership for a community of smart, curious people. **Join my free email newsletter for exclusive access** or sign up for Medium here.

*You can check my **video course** on Udemy: How to Identify, Diagnose, and Fix Memory Leaks in Web Apps.*

Programming JavaScript Technology Software Development

Software Engineering

Written by Rakia Ben Sassi

6.3K Followers · Writer for Better Programming

Follow