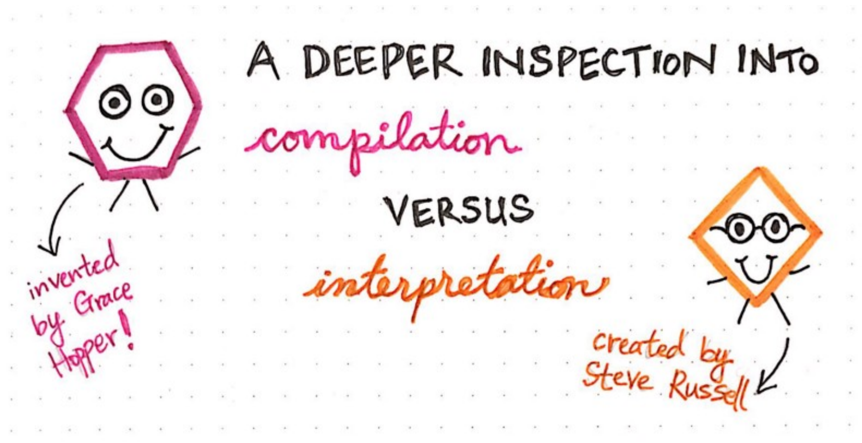
Vaidehi Joshi Follow
Writing words, writing code. Sometimes doing both at once.
Dec 13, 2017 · 15 min read

A Deeper Inspection Into Compilation And Interpretation



A deeper inspection into compilation vs. interpretation

There is perhaps nothing more satisfying than seeing the pieces of a puzzle come together. This is the case for actual puzzles, which I am pretty bad at because I always seem to lose pieces under the couch, and more metaphorical puzzles, which I am generally better at what with no pieces to lose in strange places.

The puzzle of learning is certainly one of the more complicated enigmas out there. Learning a new thing is hard because you're trying to constantly piece together ideas and construct concepts without always necessarily knowing how those parts fit into the larger whole. It can be difficult to try to wrap your head around something new when you constantly find yourself wondering what this new thing has to do with the broader picture that you're already familiar with. It's a little bit like finding a random puzzle piece and then trying to find the pieces that fit around it, but not necessarily knowing how those pieces fit into the big picture.

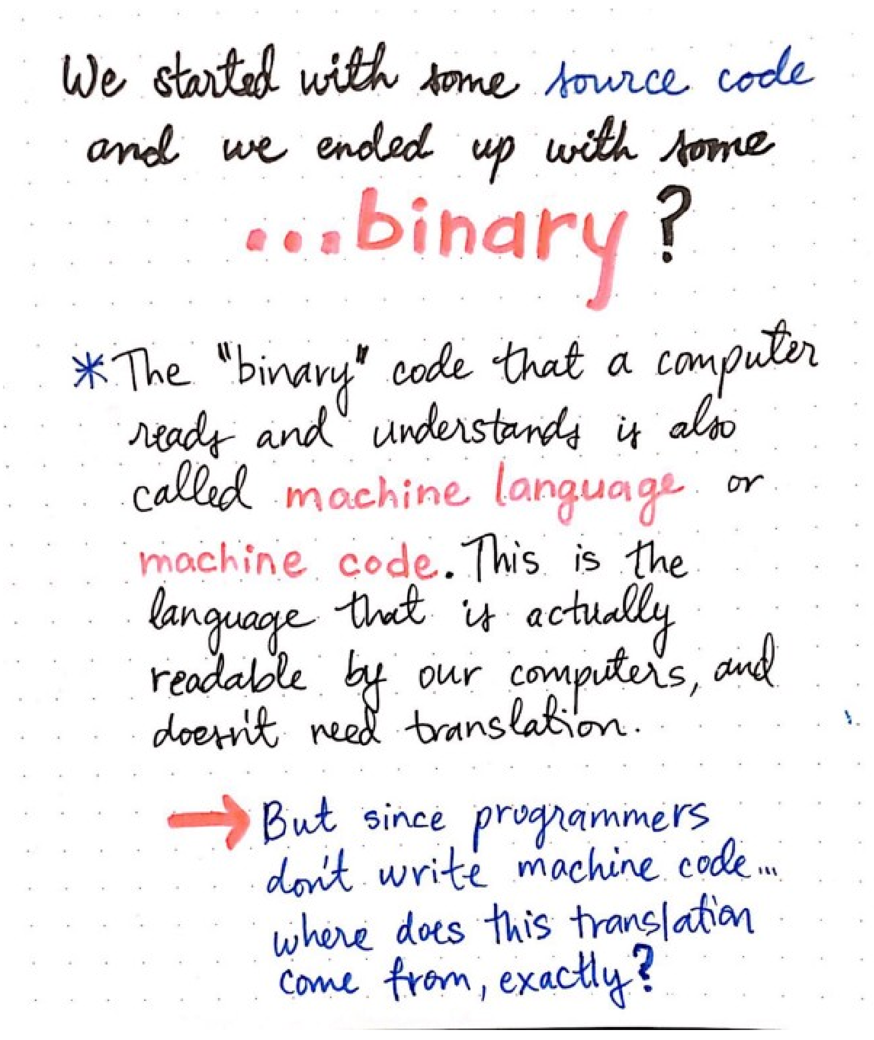
The same is true for learning a specific topic, like computer science. Oftentimes, it can feel like you're picking up little bit of information—a data structure here, an algorithm there—without always knowing how one piece connects to another. I tend to think that this is the reason that learning computer science is so hard: there just aren't very many resources that can construct the picture of the field with all of these perfectly pieces fit into one another.

However, every so often, if you really stick with a topic for long enough, you'll find that some pieces will start to come together. And as we round out this series together, it's time for that to finally happen!

A translator we know and love

When we first started this series nearly a year ago, the very first topic that we explored was something that is often thought of as the "cornerstone" of computer science: binary. We learned that, at their very core, binary is the language that every computer speaks and understands. Our machines, at the end of the day, each run on ones and zeros.

Since then, we've explored different data structures, like trees, graphs, and linked lists, as well as sorting algorithms and traversal or searching algorithms. Now it's time to finally bring it all together—or rather, bring it all *back* to binary.



We started with some source code and we ended up with some

...binary?

*The "binary" code that a computer reads and understands is also called machine language or machine code. This is the language that is actually readable by our computers, and doesn't need translation.

→ But since programmers don't write machine code... where does this translation come from, exactly?

We started with some source code and now we're here.

We've gone around the world when it comes to core concepts in computing as well as computer science. But there's one question that we haven't really answered yet, even though it's possible that we've been thinking about it this whole year: How on earth do we go from our code into the one's and zero's of our computer?

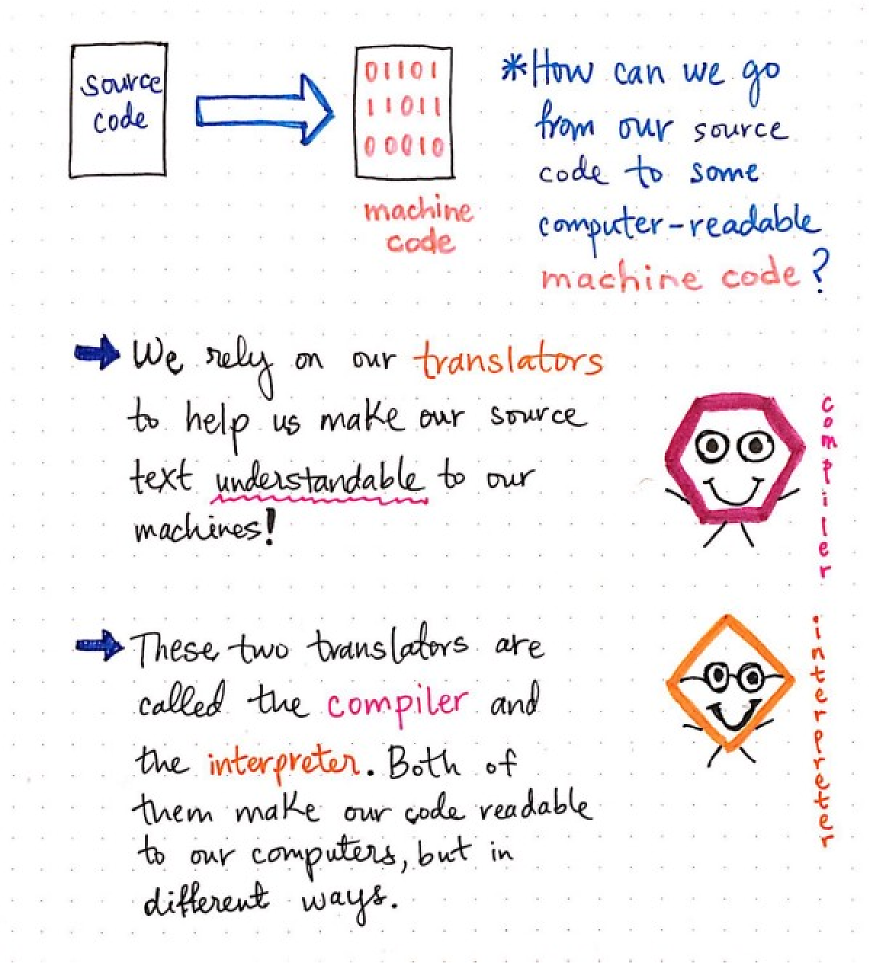
Well, before we get too far into *how* the code we write gets turned into binary, let's clarify what we really mean when we use the term binary in this context. The "binary" code that a computer reads and understands is generally referred to as ***machine language*** or ***machine code***, which is a set of instructions given to a machine and run by a its ***central processing unit*** (or ***CPU***).

An important thing to note is that there are different kinds of machine code, some of which are literally 0's and 1's, and others of which are decimals or hexadecimals (which we already know a lot about!). Regardless of which exact format a machine language is written in, it has to be fairly rudimentary because it needs to be understood by or computer. This is why machine languages are referred to as low level languages, because they need to be simplified enough to be processed by our machine's CPU, which we already know is just a bunch of switches, internally.

> *We can think of low level languages as the "mother-tongues" of our computer; machine code should be directly readable by our machine, and shouldn't ever need to be translated by them.*

But how do we go from *our* code to a machine-friendly (*machine code*) version of the exact same thing? Well, the code that we write as programmers and the machine code that a computer's processor reads are nothing more than two different types of *languages*. If we think about it, all we really need to do is translate between these two languages.

Now comes another problem: we have no idea how to translate between our code and machine code! Okay, just kidding—this isn't *really* a problem. Because we have two helpful friends who can help us out here.


source code
01101
11011
00010
machine code
*How can we go from our source code to some computer-readable machine code?
We rely on our translators to help us make our source text understandable to our machines!
compiler
These two translators are called the compiler and the interpreter. Both of them make our code readable to our computers, but in different ways.
interpreter

Translators make our source text understandable to our machines!

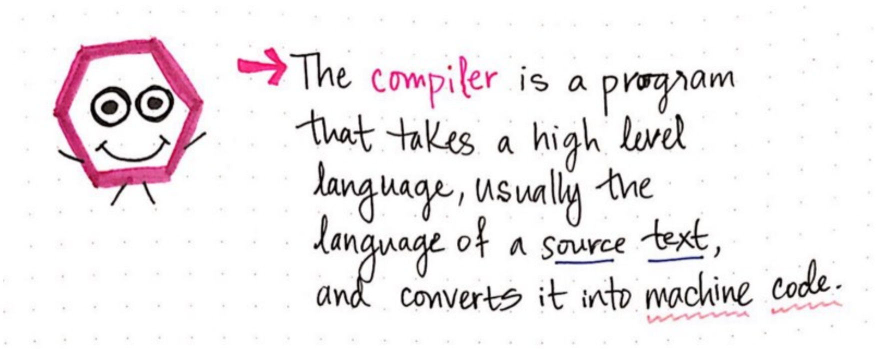
In order for us to transform our source code into some computer-readable machine code in a binary format, we'll need to rely on translators to help us make our source text understandable to our machines.

A ***translator***, which is also sometimes called a programming language processor, is nothing more than a program that translates a ***source*** language into a ***target*** language, while maintaining the logical structure of the code that it is translating.

We're already a little bit familiar with one kind of translator already, even though we might not know it just yet. Previously in this series, we've looked at the lexical and syntax analysis phases (the front end) of the compiler, and the different data structures involved in the process.

As it turns out, the compiler is a *kind* of translator! There is another translator, too, whose name often gets thrown around in conjunction with the compiler's, called the ***interpreter***. Both the compiler and the interpreter make or code readable to our computers, but in very different ways.

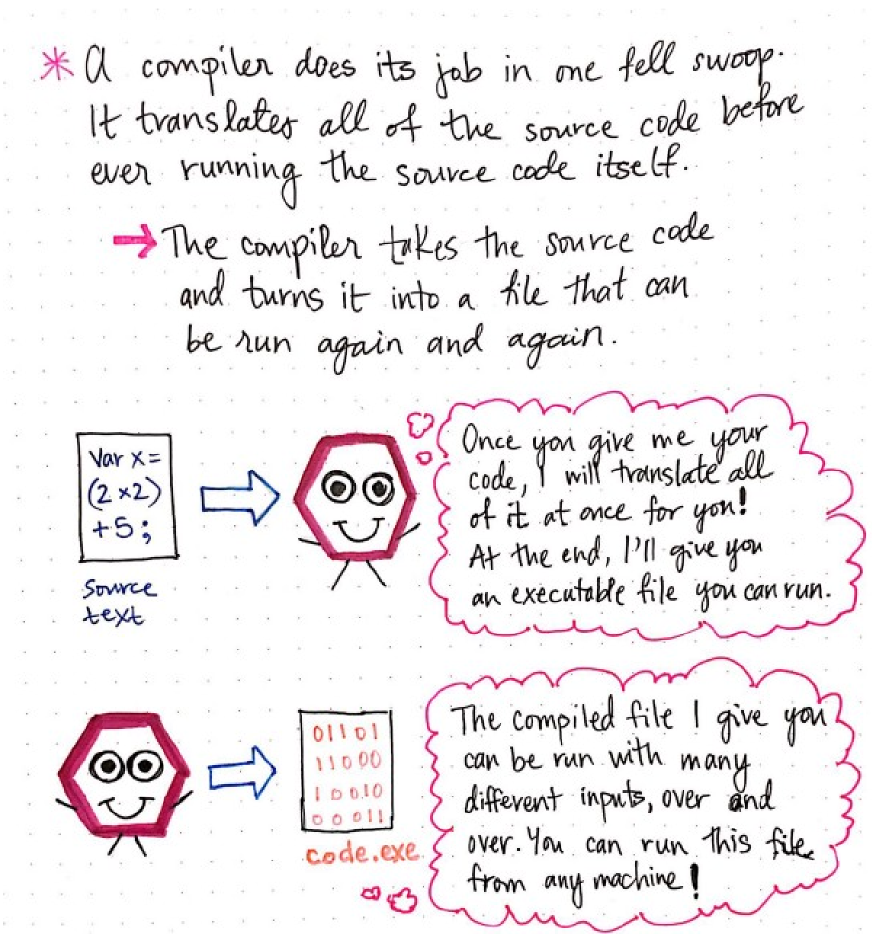
But let's first start with what we already know, first: the compiler.


The compiler is a program that takes a high level language, usually the language of a source text, and converts it into machine code.

The compiler: a definition.

The ***compiler*** is nothing more than a program that takes a high level language—the language that we write our code in—and converts it into machine code. The compiler has many moving parts to it (or rather, within it), including, potentially, a scanner, a lexer/tokenizer, and a parser. But at the end of the day, even with its complexity, it's just a program that turns our code into machine-readable code.

However, even though its job might seem simple when we put it like this, the way that a compiler does this important task is worth highlighting.


A compiler does its job in one fell swoop. It translates all of the source code before ever running the source code itself.
The compiler takes the source code and turns it into a file that can be run again and again.
Var x = (2 × 2) + 5;
Source text
Once you give me your code, I will translate all of it at once for you! At the end, I'll give you an executable file you can run.
01101
11000
10010
00011
code.exe
The compiled file I give you can be run with many different inputs, over and over. You can run this file from any machine!

How the compiler does its job.

For most cases, the compiler does the job of translating our code into machine code in one fell swoop. In other words, the compiler translates all of a programmer's source code before the source code can ever be executed or run. It takes our source code and converts it into a single file that is written in machine code. It is that very machine code file—called an ***executable*** file and often ending with an `.exe` extension—which actually allows us to run the original code that we wrote.

The most important idiosyncrasy of a compiler is the fact that it takes a source text, and translates it into machine code binary in "one shot". It returns the translated, compiled file to the programmer, who will be able to run their code via the outputted executable.

> *The executable file returned by the compiler can be run again and again once it has been translated; the*

compiler doesn't need to be around for any subsequent reruns!

Once the compiler translates all the source code into machine code, the compiler's job is done. The programmer can run the compiled code as many times as they want, with whichever inputs that they'd like to use. They can also share this compiled code with others, without ever having to share their original source code.

Grace Hopper, © TechCrunch

The concept behind this specific translator— as well as the term "compiler" itself—was coined by the illustrious ***Grace Hopper***, back in 1952, in the most interesting of circumstances.

At the time, Hopper had been working at the Eckert-Mauchly Computer Corporation, helping develop the UNIVAC I computer as a mathematician on the team. Effectively, she was working on turning mathematical code into its own language (A-0 System language).

However, she had bigger ideas. She wanted to write an entirely new programming language that would words in English, rather than the limited number of mathematical symbols. However, when she shared this idea with her colleages. They shot her down, and told her that her idea wasn't possible because "computers didn't understand English". But she was not deterred.

After three years of working on this team, Hopper had her very first working compiler. But no one believed that she had actually done it! In her biography, *Grace Hopper : Navy Admiral and Computer Pioneer*, she explains:

> *I had a running compiler and nobody would touch it. … they carefully told me, computers could only do arithmetic; they could not do programs.*

It's a good thing that Grace Hopper didn't listen to any of those nonbelievers, because she ended up continuing her work and developing one of the earliest high level programming languages, called COBOL. She also won the Presidential Medal of Freedom, among many, many, many other accomplishments.

Indeed, if she *had* listened to all of those pepole, she would have likely never taken computing to an entirely new level with her early work in constructing and designing the very first compiler. Grace Hopper's work on the first compiler laid the groundwork for the another translator that came into existence a few years later: the interpreter.

Step-by-step translation

In 1958, a few years after Grace Hopper's work on the compiler, some students at MIT were in a lab, working with an IBM 704 computer, a fairly new technology that had been introduced just four years earlier. One of these students, named Steve Russell, was working on a project called the MIT Artificial Intelligence Project, with his professor, John McCarthy.

Russell was working with the Lisp programming language at the time, and he had read a paper written by his professor on the subject. He came up with the idea to transform the `eval` function in Lisp into machine code, which set him on the path to creating the first Lisp interpreter, which was used to evaluate expressions in the language—the equivalent of running a program in Lisp at the time.

Steve Russell, © mass:werk

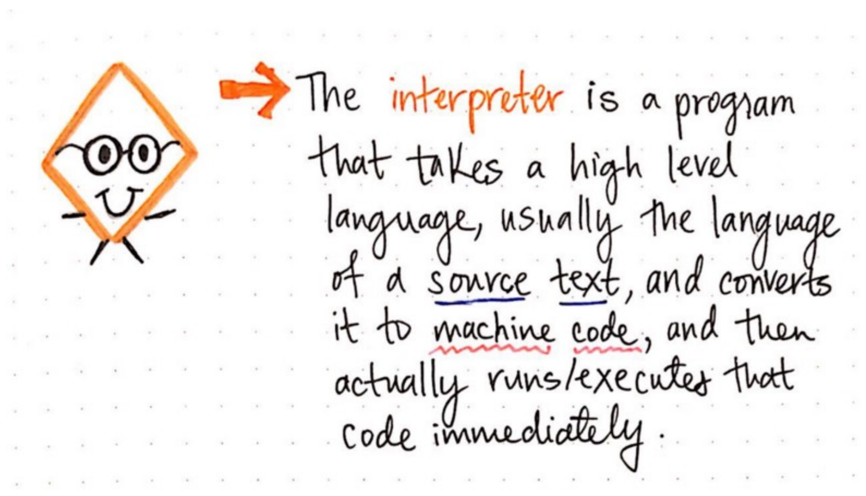
Indeed, Hopper's work directly impacted Russell's invention. The first version of the Lisp interpreter was compiled by hand. In an interview with the Computer History Museum in 2008, Russell explains how compilers played into his work at MIT:

> *And my remembrance is that John [McCarthy] sort of came in one day, in late September or October or something like, that with the universal M-expression, that is the Lisp interpreter written out as an M-expression, and we sort of looked at it and said, "Oh yeah, that'll work," and I looked at it and said, "Oh, that's just a matter of doing more hand compiling, like I've been doing. I can do that."*

> *…I got something working before Christmas, which was a useable interpreter; no garbage collector, but there weren't any big programs yet.*

Russell and his colleagues would go on to hand-compile the first two version of the Lisp interpreter. Today, most programmers wouldn't even dream of hand-compiling any of their code! Indeed, many of us interact with an interpreter and use it multiple times in application development —we just might not always be aware of it.

So, what exactly *is* an interpreter? It's about time for an official definition!

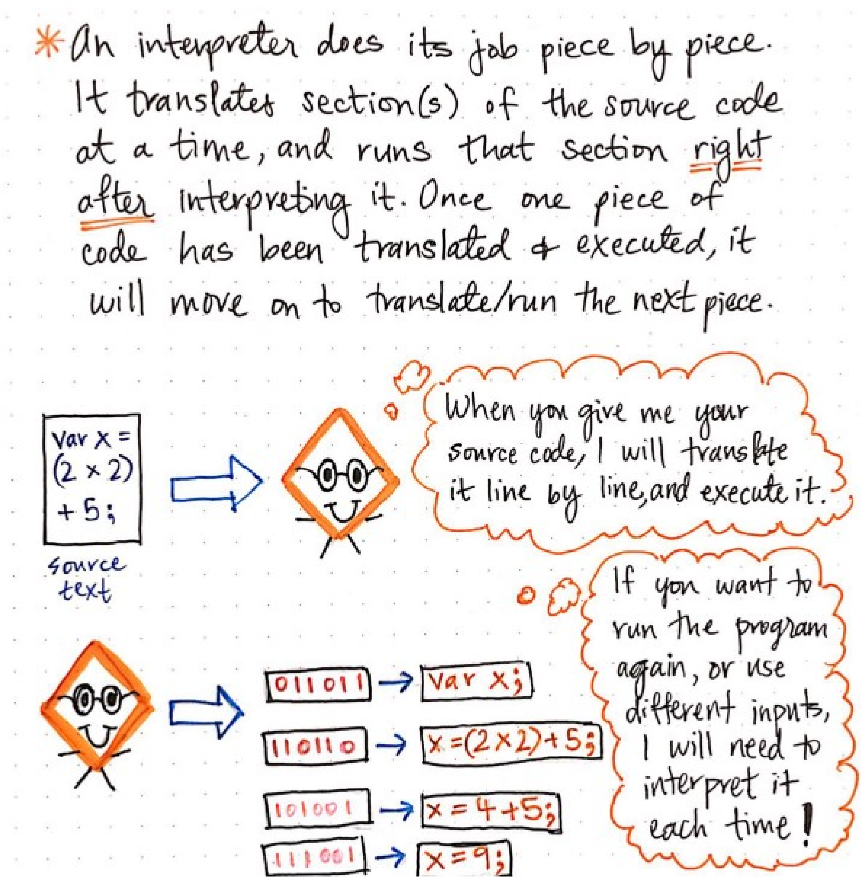

The interpreter is a program that takes a high level language, usually the language of a source text, and converts it to machine code, and then actually runs/executes that code immediately.

The interpreter: a definition.

An ***interpreter*** is also a translator, just like a compiler, in that it takes a high level language (our source text) and converts it into machine code. However, it does something slightly different: it actually runs and *executes* the code that it translates immediately (inline) as it translates.

> *We can think of an interpreter as the more "methodical" translator in the family. Rather than doing the work of translating our code into machine language in one single shot, it's far more systematic about how it works.*

An interpreter does its job piece by piece. It will translate a section of our source text at a time, rather than translating it all at once.



* An interpreter does its job piece by piece. It translates section(s) of the source code at a time, and runs that section right after interpreting it. Once one piece of code has been translated & executed, it will move on to translate/run the next piece.
var x = (2 × 2) + 5;
source text
When you give me your source code, I will translate it line by line, and execute it.
011011 → var x;
110110 → x = (2×2)+5;
101001 → x = 4+5;
111001 → x = 9;
If you want to run the program again, or use different inputs, I will need to interpret it each time!

How the interpreter does its job.

Unlike a compiler, it doesn't translate everything and hand over a file to us, the programmers, to execute. Instead, an interpreter will translate a single line/section of code at a time. Once it has finished translating one line, it will take the machine code version of it, and run it immediately.

Another way of thinking about it is that once a piece of code has been translated by the interpreter, only then can it be run. This seems fairly intuitive at first thought, because of course how can an interpreter run a line of code without knowing what it means in binary/machine code? But, if we think about it more deeply, there are other implications. Only once the interpreter finishes running one line of code successfully will it actually move onto the next line. We can imagine how this may or may not be a good thing, depending on what we're trying to do.

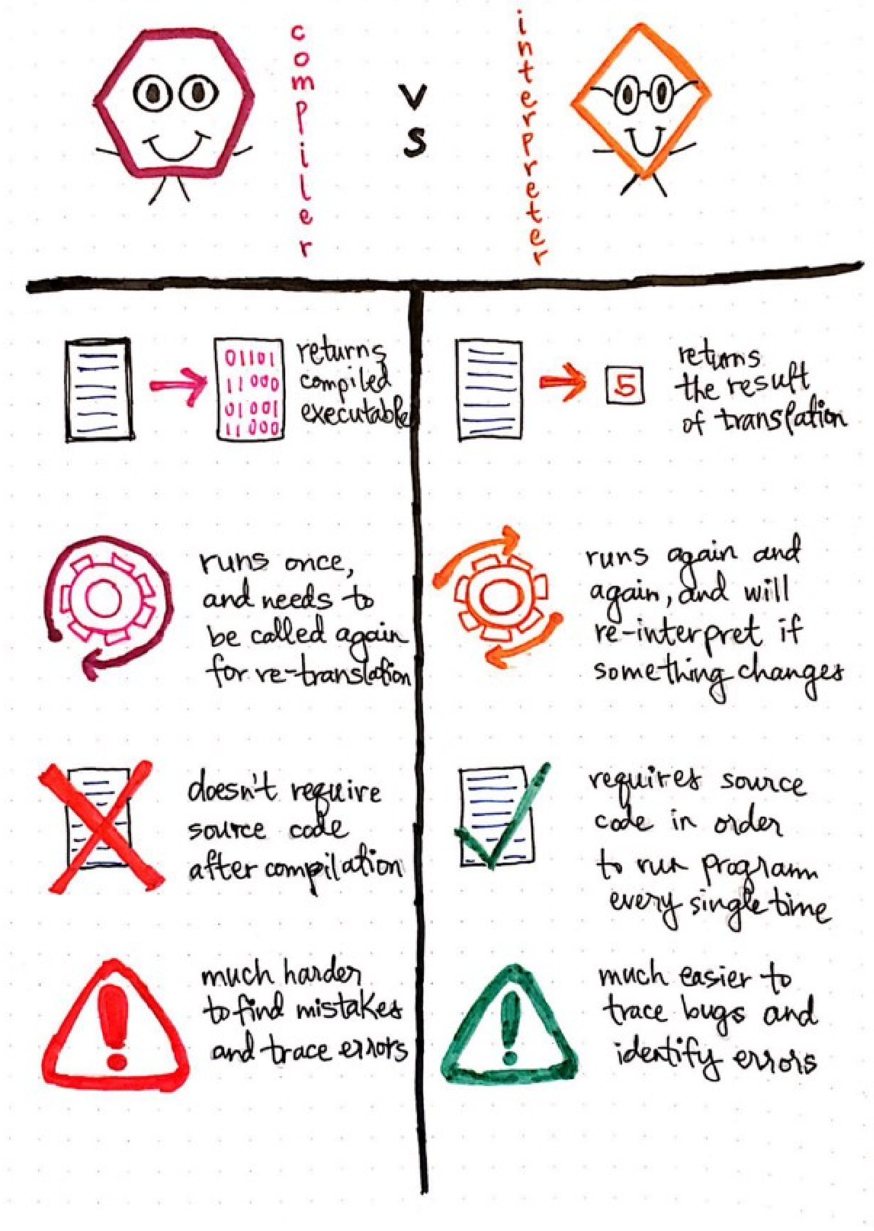
For example, imagine we want to run our program with 10 different inputs. Our interpreter would have to run our program 10 times, interpreting it line by line, for each of our inputs. However, if we made a fatal flaw in our code, our interpreter could catch it for us the moment that it happens because it has literally just tried to run our (broken) line of code after it translated it!

By now, we might start to be able to see how both the interpreter and the compiler have tradeoffs. We've seen how different tools have pros and cons to them time and again in this series, and it's probably a hallmark trait within the world of computer science. Keeping with this theme, let's weigh the fundamental differences between interpretation and compilation as translation techniques.

Two translators, both alike in dignity

The differences between interpretation and compilation and their respective translators tells us a lot about how these two programs are implemented. If we compare these two methodologies, we'll start to see how they both accomplish the same task, but in fundamentally very different ways.

The illustration below illustrates this in a more obvious way.


compiler
vs
interpreter
01101
11000
01001
11000
returns compiled executable
5
returns the result of translation
runs once, and needs to be called again for re-translation
runs again and again, and will re-interpret if something changes
doesn't require source code after compilation
requires source code in order to run program every single time
much harder to find mistakes and trace errors
much easier to trace bugs and identify errors

Compilation vs. interpretation: the tradeoffs.

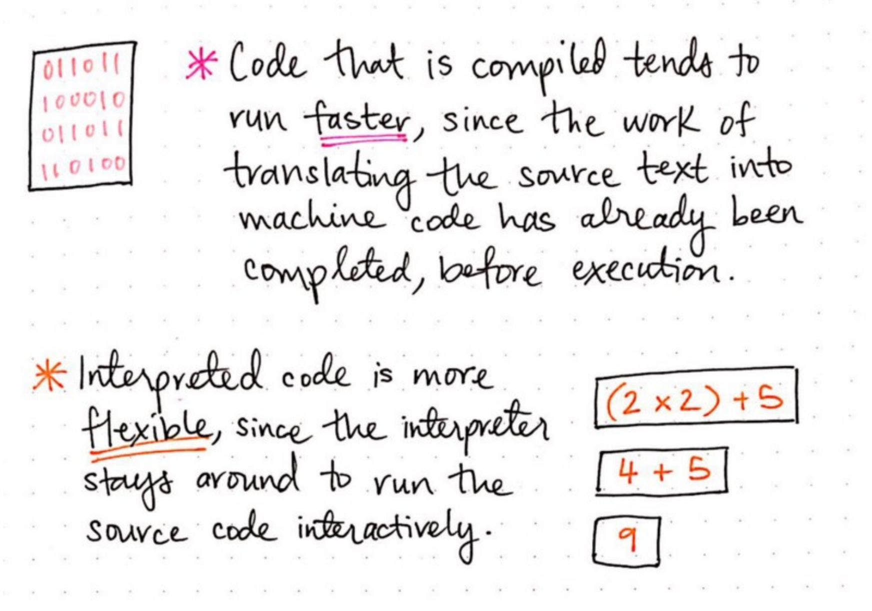
1. ***Returned result.*** While a compiler will take some source code and return a compiled, executable file, an interpreter will actually translate and then execute the source code itself, returning the result of the translation directly.

2. ***Run frequency.*** A compiler will run only once, and will need to be called again to re-translate source code if it changes. On the other hand, an interpreter will run again and reinterpret source code when it changes; the interpreter "sticks around" to continually translate.

3. ***Flexibility.*** The compiler translates the source code in one shot, which means that the source code isn't required again after compilation. However, the interpreter does require the source code in order to translate and execute the program, every single time that it is ever run.

4. ***Debugging.*** The compiler generally makes it more difficult to determine where mistakes occur in the source code, because the entire program has already been translated, and the error's location might not be easily identifyable in the machine code. However, identifying errors is easier with an interpreter because it can maintain the location of an error or bug, and surface that issue to the programmer who wrote the code.

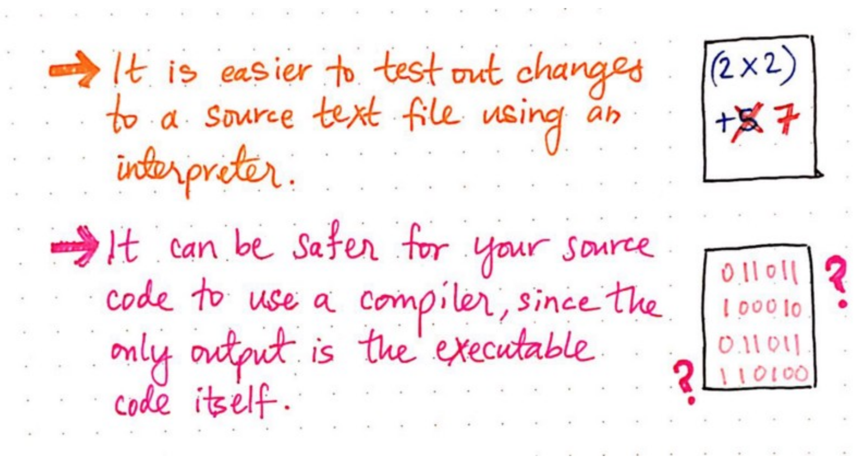
Because of these major differences, compiled code—code that is translated and run using the compilation process—tends to run a bit ***faster*** than interpreted code. This is because the work of translating the source text into machine code has already been done, before the code is ever executed.


011011
100010
011011
110100
* Code that is compiled tends to run faster, since the work of translating the source text into machine code has already been completed, before execution.
* Interpreted code is more flexible, since the interpreter stays around to run the source code interactively.
(2 x 2) + 5
4 + 5
9

Compiled code as compared to interpreted code.

On the flip side, interpreted code is far more ***flexible***, since the interpreter stays around for the course of the translation "process" to read and process our code.

Flexibility in this context means being able to change our code and being able to immediately run it afterwards. There is no need to recompile our code if we make a change; the interpreter will just pick up on that chance, and reinterpret the code, making it a much more *interactive* form of translation. Using an interpreter makes it much easier to test out small (or big!) changes in a source file.


It is easier to test out changes to a source text file using an interpreter.
(2 x 2) + ~~5~~ 7
It can be safer for your source code to use a compiler, since the only output is the executable code itself.
011011
100010
011011
110100
?
?

The benefits of compilation as compared to interpretation.

However, when it comes to interpretation, we actually need the source code in order to be able to do anything. It definitely is easier to test out changes and debug issues, but the source text has to be accessible, first and foremost. With compilation, this isn't the case. Once we have compiled our code into an executable file, we don't ever have to worry about the source code again—unless, of course, we need to recompile.

This can often make a compiler the "safer" choice, because our source code is not exposed; rather, the only output is the executable file itself, which is just 1's and 0's, and doesn't ever show anyone *how* we wrote our code or what it actually ever *said*, since it's all machine language at that point.

> *Compilation and interpretation play into our roles not just as developers of software, but also as consumers of it.*