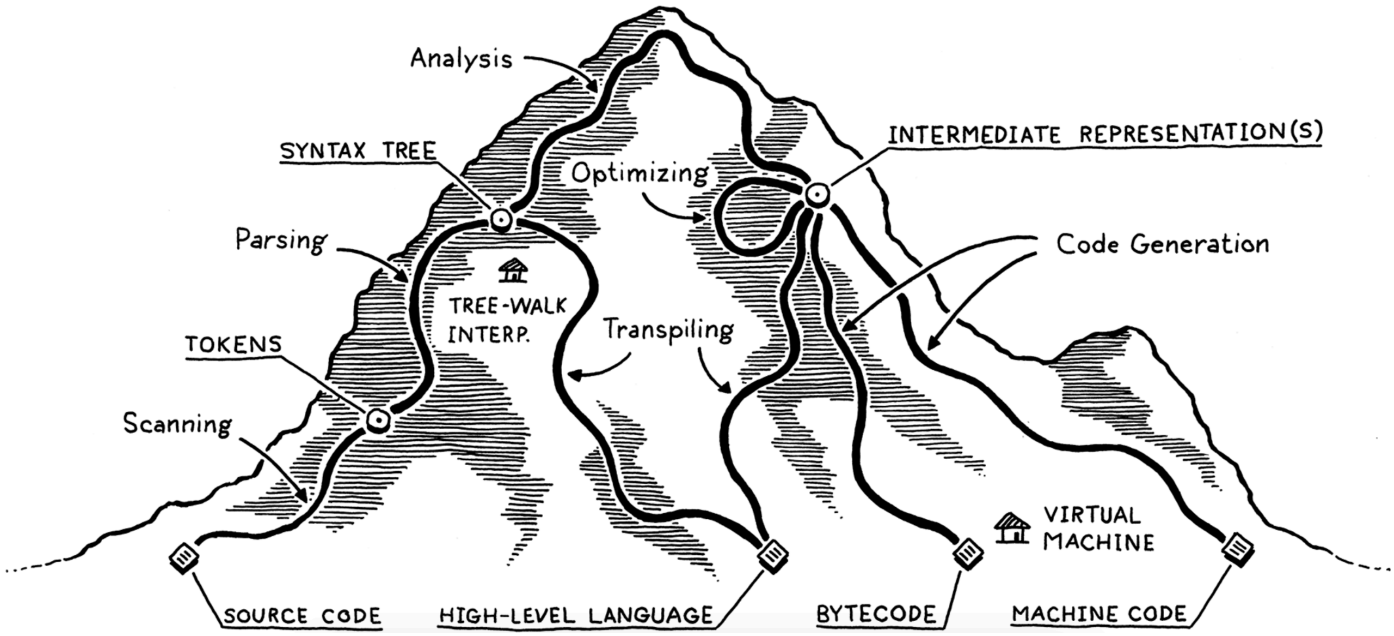
A Map of the Territory



Our journey begins with the bare text of the user's source code:

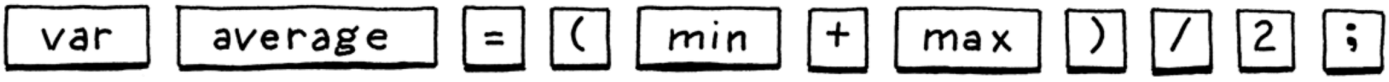
```
var average = (min + max) / 2;
```

Scanning

The first step is **scanning**, also known as **lexing**, or **lexical analysis**.

A **scanner**(or **lexer**) takes in the linear stream of characters and chunks them together into a series of something more akin to "words". In programming languages, each of these words is called a **token**. Some tokens are single characters, like (and ,. Others may be several characters long, like numbers (123), string literals ("hi!"), and identifies (min).

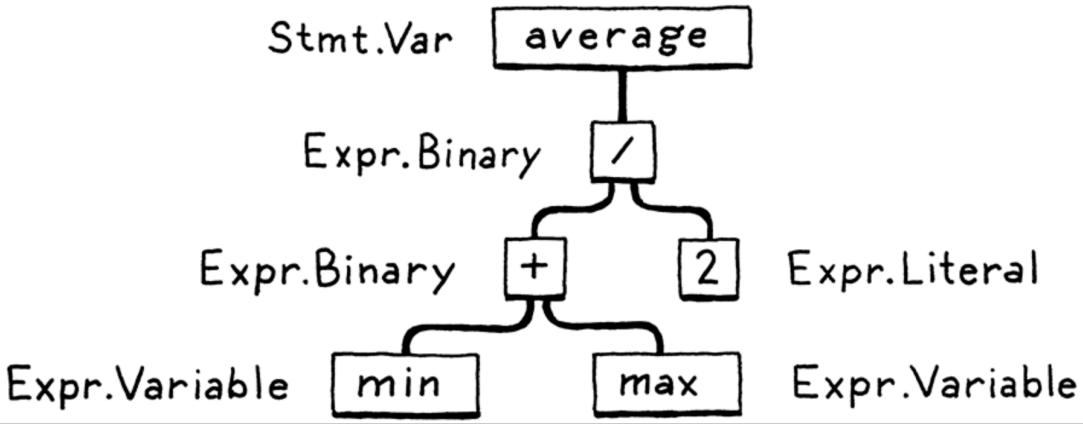
Some characters in a source file don't actually mean anything. Whitespace is often insignificant, and comments, by definition, are ignored by the language. The scanner usually discards these, leaving a clean sequence of meaningful tokens.



Parsing

The nest step is **parsing**. This is where our syntax gets a **grammar** -- the ability to compose larger expressions and statements out of smaller parts. Did you ever diagram sentences in English class? If so, you've done what a parser does, except that English has thousands and thousands of "keywords" and an overflowing cornucopia of ambiguity. Programming languages are much simpler.

A parser takes the flat sequence of token and builds a tree structure that mirrors the nested nature of the grammar. These threes have a couple of different names --**parse tree** or **abstract syntax tree** -- depending on how close to the bare syntactic structure of the source language they are. In practice, language hackers usually call them **syntax tree, ASTs**, or often just **trees**.



Parsing has a long, rich history in computer science that is closely tied to the artificial intelligence community. Many of the techniques used today to parse programming languages were originally conceived to parse *human* languages by AI researchers who were trying to get computers to talk to us.

It turns out human languages were too messy for the rigid grammars those parsers could handle, but they were a perfect fit for the simpler artificial grammars of programming languages. Alas, we flawed humans still manage to use those simple grammars incorrectly, so the parser's job also includes letting us know when we do by reporting **syntax errors**.

Static analysis

The first two **stages** are pretty similar across all implementations. Now, the individual characteristics of each language start coming into play. At this point, we know the syntactic structure of the code—things like which expressions are nested in which—but we don't know much more than that.

In an expression like `a + b`, we know we are adding `a` and `b`, but we don't know what those names refer to. Are they local variables? Global? Where are they defined?

The first bit of analysis that most languages do is called **binding** or **resolution**. For each **identifier**, we find out where that name is defined and wire the two together. This is where **scope** comes into play—the region of source code where a certain name can be used to refer to a certain declaration.

If the language is statically typed, this is when we type check. Once we know where `a` and `b` are declared, we can also figure out their types. Then if those types don't support being added to each other, we report a **type error**.

The language we'll build in this book is dynamically typed, so it will do its type checking later, at runtime.

Take a deep breath. We have attained the summit of the mountain and a sweeping view of the user's program. All this semantic insight that is visible to us from analysis needs to be stored somewhere. There are a few places we can squirrel it away:

- Often, it gets stored right back as **attributes** on the syntax tree itself—extra fields in the nodes that aren't initialized during parsing but get filled in later.
- Other times, we may store data in a lookup table off to the side. Typically, the keys to this table are identifiers—names of variables and declarations. In that case, we call it a **symbol table** and the values it associates with each key tell us what that identifier refers to.
- The most powerful bookkeeping tool is to transform the tree into an entirely new data structure that more directly expresses the semantics of the code. That's the next section.

Everything up to this point is considered the **front end** of the implementation. You might guess everything after this is the **back end**, but no. Back in the days of yore when "*front end*" and "*back end*" were coined, compilers were much simpler. Later researchers invented new phases to stuff between the two halves. Rather than discard the old terms, William Wulf and company lumped those new phases into the charming but spatially paradoxical name **middle end**.

Imtermediate representations

In the middle, the code may be stored in some **intermediate representations(IR)** that isn't tightly tied to either the source or destination forms. Instead, the IR acts as an interface between these two languages.

This lets you support multiple source languages and target platforms with less effort. Say you want to implement Pascal, C, and Fortran compilers, and you want to target x86, ARM, and, I dunno, SPARC. Normally, that means you're signing up to write *nine* full compilers: Pascal→x86, C→ARM, and every other combination.

A shared intermediate representation reduces that dramatically. You write *one* front end for each source language that produces the IR. Then *one* back end for each target architecture. Now you can mix and match those to get every combination.

There's another big reason we might want to transform the code into a form that makes the semantics more apparent . . .

Optimization

Once we understand what the user's program means, we are free to swap it out with a different program that has the *same semantics* but implements them more efficiently—we can **optimize** it.

A simple example is **constant folding**: if some expression always evaluates to the exact same value, we can do the evaluation at compile time and replace the code for the expression with its result. If the user typed in this:

```
pennyArea = 3.14159 * (0.75 / 2) * (0.75 / 2);
```

we could do all of that arithmetic in the compiler and change the code to:

```
pennyArea = 0.4417860938;
```

Optimization is a huge part of the programming language business. Many language hackers spend their entire careers here, squeezing every drop of performance they can out of their compilers to get their benchmarks a fraction of a percent faster. It can become a sort of obsession.

We're mostly going to hop over that rathole in this book. Many successful languages have surprisingly few compile-time optimizations. For example, Lua and CPython generate relatively unoptimized code, and focus most of their performance effort on the runtime.

Code generation

We have applied all of the optimizations we can think of to the user's program. The last step is converting it to a form the machine can actually run. In other words, **generating code** (or **code gen**), where "code" here usually refers to the kind of primitive assembly-like instructions a CPU runs and not the kind of "source code" a human might want to read.

Finally, we are in the **back end**, descending the other side of the mountain. From here on out, our representation of the code becomes more and more primitive, like evolution run in reverse, as we get closer to something our simple-minded machine can understand.

We have a decision to make. Do we generate instructions for a real CPU or a virtual one? If we generate real machine code, we get an executable that the OS can load directly onto the chip. Native code is lightning fast, but generating it is a lot of work. Today's architectures have piles of instructions, complex pipelines, and enough historical baggage to fill a 747's luggage bay.

Speaking the chip's language also means your compiler is tied to a specific architecture. If your compiler targets x86 machine code, it's not going to run on an ARM device. All the way back in the '60s, during the Cambrian explosion of computer architectures, that lack of portability was a real obstacle.

To get around that, hackers like Martin Richards and Niklaus Wirth, of BCPL and Pascal fame, respectively, made their compilers produce *virtual* machine code. Instead of instructions for some real chip, they produced code for a hypothetical, idealized machine. Wirth called this **p-code** for *portable*, but today, we generally call it **bytecode** because each instruction is often a single byte long.

These synthetic instructions are designed to map a little more closely to the language's semantics, and not be so tied to the peculiarities of any one computer architecture and its accumulated historical cruft. You can think of it like a dense, binary encoding of the language's low-level operations.

Virtual machine

If your compiler produces bytecode, your work isn't over once that's done. Since there is no chip that speaks that bytecode, it's your job to translate. Again, you have two options. You can write a little mini-compiler for each target architecture that converts the bytecode to native code for that machine. You still have to do work for each chip you support, but this last stage is pretty simple and you get to reuse the rest of the compiler pipeline across all of the machines you support. You're basically using your bytecode as an intermediate representation.

Or you can write a **virtual machine** (**VM**), a program that emulates a hypothetical chip supporting your virtual architecture at runtime. Running bytecode in a VM is slower than translating it to native code ahead of time because every instruction must be simulated at runtime each time it executes. In return, you get simplicity and portability. Implement your VM in, say, C, and you can run your language on any platform that has a C compiler. This is how the second interpreter we build in this book works.