Introduction 1

> *Fairy tales are more than true: not because they tell us that dragons exist, but because they tell us that dragons can be beaten.*
>
> — G.K. Chesterton by way of Neil Gaiman, *Coraline*

I'm really excited we're going on this journey together. This is a book on implementing interpreters for programming languages. It's also a book on how to design a language worth implementing. It's the book I wish I'd had when I first started getting into languages, and it's the book I've been writing in my head for nearly a decade.

To my friends and family, sorry I've been so absentminded!

In these pages, we will walk step-by-step through two complete interpreters for a full-featured language. I assume this is your first foray into languages, so I'll cover each concept and line of code you need to build a complete, usable, fast language implementation.

In order to cram two full implementations inside one book without it turning into a doorstop, this text is lighter on theory than others. As we build each piece of the system, I will introduce the history and concepts behind it. I'll try to get you familiar with the lingo so that if you ever find yourself at a cocktail party full of PL (programming language) researchers, you'll fit in.

Strangely enough, a situation I have found myself in multiple times. You wouldn't believe how much some of them can drink.

But we're mostly going to spend our brain juice getting the language up and running. This is not to say theory isn't important. Being able to reason precisely and formally about syntax and semantics is a vital skill when working on a language. But, personally, I learn best by doing. It's hard for me to wade through paragraphs full of abstract concepts and really absorb them. But if I've coded something, run it, and debugged it, then I *get* it.

Static type systems in particular require rigorous formal reasoning. Hacking on a type system has the same feel as proving a theorem in mathematics.

It turns out this is no coincidence. In the

That's my goal for you. I want you to come away with a solid intuition of how a real language lives and breathes. My hope is that when you read other, more theoretical books later, the concepts there will firmly stick in your mind, adhered to this tangible substrate.

early half of last century, Haskell Curry and William Alvin Howard showed that they are two sides of the same coin: the Curry-Howard isomorphism.

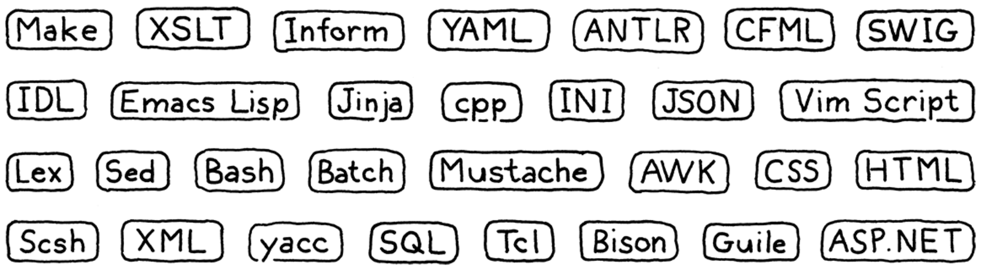
Why Learn This Stuff? 1.1

Every introduction to every compiler book seems to have this section. I don't know what it is about programming languages that causes such existential doubt. I don't think ornithology books worry about justifying their existence. They assume the reader loves birds and start teaching.

But programming languages are a little different. I suppose it is true that the odds of any of us creating a broadly successful, general-purpose programming language are slim. The designers of the world's widely used languages could fit in a Volkswagen bus, even without putting the pop-top camper up. If joining that elite group was the *only* reason to learn languages, it would be hard to justify. Fortunately, it isn't.

Little languages are everywhere 1.1.1

For every successful general-purpose language, there are a thousand successful niche ones. We used to call them "little languages", but inflation in the jargon economy led to the name "domain-specific languages". These are pidgins tailor-built to a specific task. Think application scripting languages, template engines, markup formats, and configuration files.



A random selection of some little languages you might run into.

Almost every large software project needs a handful of these. When you can, it's good to reuse an existing one instead of rolling your own. Once you factor in documentation, debuggers, editor support, syntax highlighting, and all of the other trappings, doing it yourself becomes a tall order.

But there's still a good chance you'll find yourself needing to whip up a parser or other tool when there isn't an existing library that fits your needs. Even when you are reusing some existing implementation, you'll inevitably end up needing to debug and maintain it and poke around in its guts.

Languages are great exercise 1.1.2

Long distance runners sometimes train with weights strapped to their ankles or at high altitudes where the atmosphere is thin. When they later unburden themselves, the new relative ease of light limbs and oxygen-rich air enables them to run farther and faster.

Implementing a language is a real test of programming skill. The code is complex and performance critical. You must master recursion, dynamic arrays, trees, graphs, and hash tables. You probably use hash tables at least in your day-to-day programming, but do you *really* understand them? Well, after we've crafted our own from scratch, I guarantee you will.

While I intend to show you that an interpreter isn't as daunting as you might believe, implementing one well is still a challenge. Rise to it, and you'll come away a stronger programmer, and smarter about how you use data structures and algorithms in your day job.

One more reason 1.1.3

This last reason is hard for me to admit, because it's so close to my heart. Ever since I learned to program as a kid, I felt there was something magical about languages. When I first tapped out BASIC programs one key at a time I couldn't conceive how BASIC *itself* was made.

Later, the mixture of awe and terror on my college friends' faces when talking about their compilers class was enough to convince me language hackers were a different breed of human—some sort of wizards granted privileged access to arcane arts.

It's a charming image, but it has a darker side. *I* didn't feel like a wizard, so I was left thinking I lacked some inborn quality necessary to join the cabal. Though I've been fascinated by languages ever since I doodled made-up keywords in my school notebook, it took me decades to muster the courage to try to really learn them. That "magical" quality, that sense of exclusivity, excluded *me*.

When I did finally start cobbling together my own little interpreters, I quickly learned that, of course, there is no magic at all. It's just code, and the people who hack on languages are just people.

And its practitioners don't hesitate to play up this image. Two of the seminal texts on programming languages feature a dragon and a wizard on their covers.

There *are* a few techniques you don't often encounter outside of languages, and some parts are a little difficult. But not more difficult than other obstacles you've overcome. My hope is that if you've felt intimidated by languages and this book helps you overcome that fear, maybe I'll leave you just a tiny bit braver than you were before.

And, who knows, maybe you *will* make the next great language. Someone has to.

How the Book Is Organized 1.2

This book is broken into three parts. You're reading the first one now. It's a couple of chapters to get you oriented, teach you some of the lingo that language hackers use, and introduce you to Lox, the language we'll be implementing.

Each of the other two parts builds one complete Lox interpreter. Within those parts, each chapter is structured the same way. The chapter takes a single language feature, teaches you the concepts behind it, and walks you through an implementation.

It took a good bit of trial and error on my part, but I managed to carve up the two interpreters into chapter-sized chunks that build on the previous chapters but require nothing from later ones. From the very first chapter, you'll have a working program you can run and play with. With each passing chapter, it grows increasingly full-featured until you eventually have a complete language.

Aside from copious, scintillating English prose, chapters have a few other delightful facets:

The code 1.2.1

We're about *crafting* interpreters, so this book contains real code. Every single line of code needed is included, and each snippet tells you where to insert it in your ever-growing implementation.

Many other language books and language implementations use tools like Lex and Yacc, so-called **compiler-compilers**, that automatically generate some of the source files for an implementation from some higher-level description. There are pros and cons to tools like those, and strong opinions — some might say religious convictions — on both sides.

We will abstain from using them here. I want to ensure there are no dark corners where magic and confusion can hide, so we'll write everything by hand. As you'll see, it's not as bad as it sounds, and it means you really will understand each line of code and how both interpreters work.

A book has different constraints from the "real world" and so the coding style

Yacc is a tool that takes in a grammar file and produces a source file for a compiler, so it's sort of like a "compiler" that outputs a compiler, which is where we get the term "compiler-compiler".

Yacc wasn't the first of its ilk, which is why here might not always reflect the best way to write maintainable production software. If I seem a little cavalier about, say, omitting `private` or declaring a global variable, understand I do so to keep the code easier on your eyes. The pages here aren't as wide as your IDE and every character counts.

Also, the code doesn't have many comments. That's because each handful of lines is surrounded by several paragraphs of honest-to-God prose explaining it. When you write a book to accompany your program, you are welcome to omit comments too. Otherwise, you should probably use `//` a little more than I do.

While the book contains every line of code and teaches what each means, it does not describe the machinery needed to compile and run the interpreter. I assume you can slap together a makefile or a project in your IDE of choice in order to get the code to run. Those kinds of instructions get out of date quickly, and I want this book to age like XO brandy, not backyard hooch.

it's named "Yacc" — *Yet Another* Compiler-Compiler. A later similar tool is Bison, named as a pun on the pronunciation of Yacc like "yak".

If you find all of these little self-references and puns charming and fun, you'll fit right in here. If not, well, maybe the language nerd sense of humor is an acquired taste.

Snippets 1.2.2

Since the book contains literally every line of code needed for the implementations, the snippets are quite precise. Also, because I try to keep the program in a runnable state even when major features are missing, sometimes we add temporary code that gets replaced in later snippets.

A snippet with all the bells and whistles looks like this:

```
      default:
        if (isDigit(c)) {
          number();
        } else {
          Lox.error(line, "Unexpected character.");
        }
        break;
```

<< lox/Scanner.java
in scanToken()
replace 1 line

In the center, you have the new code to add. It may have a few faded out lines above or below to show where it goes in the existing surrounding code. There is also a little blurb telling you in which file and where to place the snippet. If that blurb says "replace _ lines", there is some existing code between the faded lines that you need to remove and replace with the new snippet.

Asides 1.2.3

Asides contain biographical sketches, historical background, references to related topics, and suggestions of other areas to explore. There's nothing that you *need* to know in them to understand later parts of the book, so you can skip them if you want. I won't judge you, but I might be a little sad.

Well, some asides do, at least. Most of them are just dumb jokes and amateurish drawings.

Challenges 1.2.4

Each chapter ends with a few exercises. Unlike textbook problem sets, which tend to review material you already covered, these are to help you learn *more* than what's in the chapter. They force you to step off the guided path and explore on your own. They will make you research other languages, figure out how to implement features, or otherwise get you out of your comfort zone.

Vanquish the challenges and you'll come away with a broader understanding and possibly a few bumps and scrapes. Or skip them if you want to stay inside the comfy confines of the tour bus. It's your book.

A word of warning: the challenges often ask you to make changes to the interpreter you're building. You'll want to implement those in a copy of your code. The later chapters assume your interpreter is in a pristine ("unchallenged"?) state.

Design notes 1.2.5

Most "programming language" books are strictly programming language *implementation* books. They rarely discuss how one might happen to *design* the language being implemented. Implementation is fun because it is so precisely defined. We programmers seem to have an affinity for things that are black and white, ones and zeroes.

I know a lot of language hackers whose careers are based on this. You slide a language spec under their door, wait a few months, and code and benchmark results come out.

Personally, I think the world needs only so many implementations of FORTRAN 77. At some point, you find yourself designing a *new* language. Once you start playing *that* game, then the softer, human side of the equation becomes paramount. Things like which features are easy to learn, how to balance innovation and familiarity, what syntax is more readable and to whom.

Hopefully your new language doesn't hardcode assumptions about the width of a punched card into its grammar.

All of that stuff profoundly affects the success of your new language. I want your language to succeed, so in some chapters I end with a "design note", a little essay on some corner of the human aspect of programming languages. I'm no expert on this — I don't know if anyone really is — so take these with a large pinch of salt. That should make them tastier food for thought, which is my main aim.

The First Interpreter 1.3

We'll write our first interpreter, jlox, in Java. The focus is on *concepts*. We'll write the simplest, cleanest code we can to correctly implement the semantics of the language. This will get us comfortable with the basic techniques and also hone our understanding of exactly how the language is supposed to behave.

Java is a great language for this. It's high level enough that we don't get overwhelmed by fiddly implementation details, but it's still pretty explicit. Unlike in scripting languages, there tends to be less complex machinery hiding under the hood, and you've got static types to see what data structures you're working with.

The book uses Java and C, but readers have ported the code to many other languages. If the languages I picked aren't your bag, take a look at those.

I also chose Java specifically because it is an object-oriented language. That paradigm swept the programming world in the '90s and is now the dominant way of thinking for millions of programmers. Odds are good you're already used to organizing code into classes and methods, so we'll keep you in that comfort zone.

While academic language folks sometimes look down on object-oriented languages, the reality is that they are widely used even for language work. GCC and LLVM are written in C++, as are most JavaScript virtual machines. Object-oriented languages are ubiquitous, and the tools and compilers *for* a language are often written *in* the same language.

A compiler reads files in one language, translates them, and outputs files in another language. You can implement a compiler in any language, including the same language it compiles, a process called **self-hosting**.

You can't compile your compiler using itself yet, but if you have another compiler for your language written in some other language, you use *that* one to compile your compiler once. Now you can use the compiled version of your own compiler to compile future versions of itself, and you can discard the original one compiled from the other compiler. This is called **bootstrapping**, from the image of pulling yourself up by your own bootstraps.

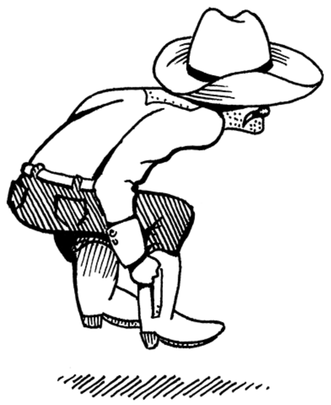
And, finally, Java is hugely popular. That means there's a good chance you already know it, so there's less for you to learn to get going in the book. If you aren't that familiar with Java, don't freak out. I try to stick to a fairly minimal subset of it. I use the diamond operator from Java 7 to make things a little more terse, but that's about it as far as "advanced" features go. If you know another object-oriented language, like C# or C++, you can muddle through.

By the end of part II, we'll have a simple, readable implementation. It's not very fast, but it's correct. However, we are only able to accomplish that by building on the Java virtual machine's own runtime facilities. We want to learn how Java *itself* implements those things.

The Second Interpreter 1.4

So in the next part, we start all over again, but this time in C. C is the perfect language for understanding how an implementation *really* works, all the way down to the bytes in memory and the code flowing through the CPU.

A big reason that we're using C is so I can show you things C is particularly good at, but that *does* mean you'll need to be pretty comfortable with it. You don't have to be the reincarnation of Dennis Ritchie, but you shouldn't be spooked by pointers either.



If you aren't there yet, pick up an introductory book on C and chew through it, then come back here when you're done. In return, you'll come away from this book an even stronger C programmer. That's useful given how many language implementations are written in C: Lua, CPython, and Ruby's MRI, to name a few.

In our C interpreter, clox, we are forced to implement for ourselves all the things Java gave us for free. We'll write our own dynamic array and hash table. We'll decide how objects are represented in memory, and build a garbage collector to reclaim them.

I pronounce the name like "sea-locks", but you can say it "clocks" or even "cloch", where you pronounce the "x" like the Greeks do if it makes you happy.

Our Java implementation was focused on being correct. Now that we have that down, we'll turn to also being *fast*. Our C interpreter will contain a compiler that translates Lox to an efficient bytecode representation (don't worry, I'll get into what that means soon), which it then executes. This is the same technique used by implementations of Lua, Python, Ruby, PHP, and many other successful languages.

Did you think this was just an interpreter book? It's a compiler book as well. Two for the price of one!

We'll even try our hand at benchmarking and optimization. By the end, we'll have a robust, accurate, fast interpreter for our language, able to keep up with other professional caliber implementations out there. Not bad for one book and a few thousand lines of code.

CHALLENGES

1. There are at least six domain-specific languages used in the little system I cobbled together to write and publish this book. What are they?

2. Get a "Hello, world!" program written and running in Java. Set up whatever makefiles or IDE projects you need to get it working. If you have a debugger, get comfortable with it and step through your program as it runs.

3. Do the same thing for C. To get some practice with pointers, define a doubly linked list of heap-allocated strings. Write functions to insert, find, and delete items from it. Test them.

DESIGN NOTE: WHAT'S IN A NAME?

One of the hardest challenges in writing this book was coming up with a name for the language it implements. I went through *pages* of candidates before I found one that worked. As you'll discover on the first day you start building your own language, naming is deviously hard. A good name satisfies a few criteria:

1. **It isn't in use.** You can run into all sorts of trouble, legal and social, if you inadvertently step on someone else's name.

2. **It's easy to pronounce.** If things go well, hordes of people will be saying and writing your language's name. Anything longer than a couple of syllables or a handful of letters will annoy them to no end.

3. **It's distinct enough to search for.** People will Google your language's name to learn about it, so you want a word that's rare enough that most results point to your docs. Though, with the amount of AI search engines are packing today, that's less of an issue. Still, you won't be doing your users any favors if you name your language "for".

4. **It doesn't have negative connotations across a number of cultures.** This is hard to be on guard for, but it's worth considering. The designer of Nimrod ended up renaming his language to "Nim" because too many people remember that Bugs Bunny used "Nimrod" as an insult. (Bugs was using it ironically.)

If your potential name makes it through that gauntlet, keep it. Don't get hung up on trying to find an appellation that captures the quintessence of your language. If the names of the world's other successful languages teach us anything, it's that the name doesn't matter much. All you need is a reasonably unique token.

NEXT CHAPTER: "A MAP OF THE TERRITORY" →