#1. 前言 Introduction

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

童话故事是无比真实的：不是因为它告诉我们龙的存在，而是因为它告诉我们龙可以被击败。

> 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 had when I first started getting into languages, and it's the book I've been writing in my head for nearly a decade.

我真的很兴奋我们能一起踏上这段旅程。这是一本关于为编程语言实现解释器的书。它也是一本关于如何设计一种值得实现的语言的书。我刚开始接触编程语言的时候就希望我可以写出这本书，这本书我在脑子里已经写了将近十年了。

> 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.

在本书中，我们将一步一步地介绍一种功能齐全的语言的两个完整的解释器实现。我假设这是您第一次涉足编程语言，因此我将介绍构建一个完整、可用、快速的语言所需的每个概念和代码。

> 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 in a cocktail party full of PL (programming language) researchers, you'll fit in.

为了在一本书中塞进两个完整的实现，而且避免这变成一个门槛，本文在理论上比其他文章更轻。在构建系统的每个模块时，我将介绍它背后的历史和概念。我会尽力让您熟悉这些行话，即便您在充满PL（编程语言）研究人员的鸡尾酒会中，也能快速融入其中。

> 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.

但我们主要还是要花费精力让这门语言运转起来。这并不是说理论不重要。在学习一门语言时，能够对语法和语义进行精确而公式化的推理[1]是一项至关重要的技能。但是，就我个人而言，我在实践中学习效果最好。对我来说，要深入阅读那些充满抽象概念的段落并真正理解它们太难了。但是，如果我（根据理论）编写了代码，运行并调试完成，那么我就明白了。

> 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.

这就是我对您的期望。我想让你们直观地理解一门真正的语言是如何生活和呼吸的。我的希望是，当你以后阅读其他理论性更强的书籍时，这些概念会牢牢地留在你的脑海中，依附于这个有形的基础之上。

#1.1 Why Learn This Stuff?

#1.1 为什么要学习这些?

> 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.

每一本编译器相关书籍的前言似乎都有这一节。我不知道为什么编程语言会引起这种存在性的怀疑。我认为鸟类学书籍作者不会担心证明它们的存在。他们假设读者喜欢鸟，然后就开始讲授内容。

> 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.

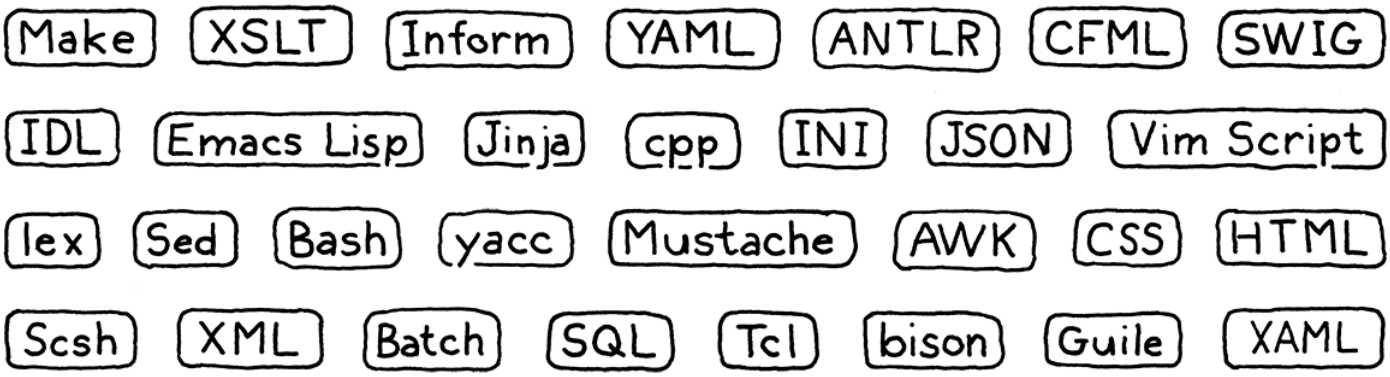
但是编程语言有一点不同。我认为，对我们中的任何一个人来说，能够创建一种广泛成功的通用编程语言的可能性都很小，这是事实。设计世界通用语言的设计师们，一辆汽车就能装得下。如果加入这个精英群体是学习语言的唯一原因，那么就很难证明其合理性。幸运的是，事实并非如此。

#1.1.1 Little languages are everywhere

#1.1.1 小型语言无处不在

> 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 today 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.

对于每一种成功的通用语言，都有上千种成功的小众语言。我们过去称它们为“小语言”，但术语泛滥的今天它们有了“领域特定语言（即DSL）”的名称。这些是为特定任务量身定做的洋泾浜语言[2]，如应用程序脚本语言、模板引擎、标记格式和配置文件。



> 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.

几乎每个大型软件项目都需要一些这样的工具。如果可以的话，最好重用现有的工具，而不是自己动手实现。一旦考虑到文档、调试器、编辑器支持、语法高亮显示和所有其他可能的障碍，自己实现就成了一项艰巨的任务。

> But there’s still a good chance you’ll find yourself needing to whip up a parser or something 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.

但是，当现有的库不能满足您的需要时，您仍然很有可能发现自己需要一个解析器或其他东西。即使当您重用一些现有的实现时，您也不可避免地需要调试和维护，并在其内部进行探索。

> #1.1.2 Languages are great exercise

#1.1.2 语言是很好的锻炼

> 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.

长跑运动员有时会在脚踝上绑上重物，或者在空气稀薄的高海拔地区进行训练。当他们卸下自己的负担以后，轻便的肢体和富氧的空气带来了新的相对舒适度，使它们可以跑得更快，更远。

> 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 how well do you *really* understand them? Well, after we've crafted our own from scratch, I guarantee you will.

实现一门语言是对编程技能的真正考验。代码很复杂，而性能很关键。您必须掌握递归、动态数组、树、图和哈希表。您在日常编程中至少使用过哈希表，但您对它们的理解程度有多高呢?嗯，等我们从头完成我们的作品之后，我相信您会理解的。

> 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.

虽然我想说明解释器并不像您想的那样令人生畏，但实现一个好的解释器仍然是一个挑战。学会了它，您就会成为一个更强大的程序员，并且在日常工作中也能更加聪明地使用数据结构和算法。

> #### 1.1.3 One more reason

1.1.3 另一个原因

> 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.

这最后一个原因我很难承认，因为它是很私密的理由。自从我小时候学会编程以来，我就觉得语言有种神奇的力量。当我第一次一个键一个键地输入BASIC程序时，我无法想象BASIC语言本身是如何制作出来的。

> 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.

后来，当我的大学朋友们谈论他们的编译器课程时，脸上那种既敬畏又恐惧的表情足以让我相信，语言黑客是另一种人，某种获得了通向神秘艺术的特权的巫师。

> 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 in-born 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*.

这是一个迷人的形象，但它也有黑暗的一面。我感觉自己不像个巫师，所以我认为自己缺乏加入秘社所需的先天品质。 尽管自从我在学校笔记本上拼写关键词以来，我一直对语言着迷，但我花了数十年的时间鼓起勇气尝试真正地学习它们。那种 “神奇”的品质，那种排他性的感觉，将我挡在门外。

> 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.

当我最终开始拼凑我自己的小编译器时，我很快意识到，根本就没有魔法。它只是代码，而那些掌握语言的人也只是人。

> 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.

有一些技巧您在语言之外不会经常遇到，而且有些部分有点难。但不会比您克服的其他障碍更困难。我希望，如果您对语言感到害怕，而这本书能帮助您克服这种恐惧，也许我会让您比以前更勇敢一点。

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

而且，说不准，你也许会创造出下一个伟大的语言，毕竟总要有人做。

#1.2 How the Book is Organized

#1.2 本书的组织方式

> 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.

这本书分为三个部分。您现在正在读的是第一部分。这部分用了几章来让您进入状态，教您一些语言黑客使用的行话，并向您介绍我们将要实现的语言Lox。

> 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 through an implementation.

其他两个部分则分别构建一个完整的Lox解释器。在这些部分中，每个章节的结构都是相同的。 每一章节挑选一个语言功能点，教您背后对应的概念，并逐步介绍实现方法。

> 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.

我花了不少时间去试错，但我还是成功地把这两个解释器按照章节分成了一些小块，每一小块的内容都会建立在前面几章的基础上，但不需要后续章节的知识。从第一章开始，你就会有一个可以运行和使用的工作程序。随着章节的推移，它的功能越来越丰富，直到你最终拥有一门完整的语言。

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

除了大量妙趣横生的英文段落，章节中还会包含一些其它的惊喜：

> #### 1.2.1 The code

1.2.1 代码

> 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.

本书是关于制作解释器的，所以其中会包含真正的代码。所需要的每一行代码都需要包含在内，而且每个代码片段都会告知您需要插入到实现代码中的 什么位置。

> 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.

许多其他的语言书籍和语言实现都使用Lex和Yacc[3]这样的工具，也就是所谓的**编译器-编译器**，可以从一些更高层次的（语法）描述中自动生成一些实现的源文件。这些工具有利有弊，而且双方都有强烈的主张--有些人可能将其说成是信仰。

> 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.

我们这里不会使用这些工具。我想确保魔法和困惑不会藏在黑暗的角落，所以我们会选择手写所有代码。正如您将看到的，这并没有听起来那么糟糕，因为

这意味着您将真正理解每一行代码以及两种解释器的工作方式。

> A book has different constraints from the “real world” and so the coding style 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.

为了写书，书中代码和“真实世界”的代码是有区别的，因此这里的代码风格可能并不是可维护生产软件的最佳方式。可能我的某些写法是不太准确的，比如省略*private*或者声明全局变量，请理解我这样做是为了让您更容易看懂代码。书页不像IDE窗口那么宽，所以每一个字符都很珍贵。

> 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.

另外，代码也不会有太多的注释。这是因为每一部分代码前后，都使用了一些真的很简洁的文字来对其进行解释。当你写一本书来配合你的程序时，欢迎你也省略注释。否则，你可能应该比我使用更多的*//*。

> 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.

虽然这本书包含了每一行代码，并教授了每一行代码的含义，但它没有描述编译和运行解释器所需的机制。我假设您可以在IDE中选择一个makefile或一个项

目导入，以使代码运行。 这类说明很快就会过时，我希望这本书能像XO白兰地一样醇久，而不是像家酿酒（一样易过期）。

> #1.2.2 Snippets

#1.2.2 片段

> 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.

因为这本书包含了实现所需的每一行代码，所以代码片段相当精确。此外，即使是在缺少主要功能的时候，我也尝试将程序保持在可运行状态。因此我们有时会添加临时代码，这些代码将在以后的代码段中替换。

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

一个完整的代码片段可能如下所示：

```
      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.

中间是要添加的新代码。这部分代码的上面或下面可能有一些淡出的行，以显示它在周围代码中的位置。还会附有一小段介绍，告诉您在哪个文件中以及在哪里放置代码片段。如果简介说要“replace x lines”，表明在浅色的行之间有一些现有的代码需要删除，并替换为新的代码片段。

> #1.2.3 Asides

#1.2.3 题外话

> 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.

题外话中包含传记简介、历史背景、对相关主题的引用以及对其他要探索的领域的建议。 您无需深入了解就可以理解本书的后续部分，因此可以根据需要跳过它们。 我不会批评你，但我可能会有些难过。【注：由于排版原因，在翻译的时候，将有用的旁白信息作为脚注附在章节之后】

> #1.2.4 Challenge

#1.2.4 挑战

> 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.

每章结尾都会有一些练习题。 不像教科书中的习题集那样用于回顾已讲述的内容，这些习题是为了帮助您学习更多的知识，而不仅仅是本章中的内容。 它们会迫使您走出文章指出的路线，自行探索。 它们将要求您研究其他语言，弄清楚如何实现功能，换句话说，就是使您走出舒适区。

> 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.

克服挑战，您将获得更广泛的理解，也可能遇到一些挫折。 如果您想留在旅游巴士的舒适区内，也可以跳过它们。 都随你便[4]。

> #### #1.2.5 Design notes

#1.2.5 设计笔记

> 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.

大多数编程语言书籍都是严格意义上的编程语言*实现*书籍。他们很少讨论如何*设计*正在实现的语言。实现之所以有趣，是因为它的定义是很精确的。我们程序员似乎很喜欢黑白、1和0这样的事物[5]。

> Personally, I think the world only needs 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 what features are easy to learn, how to balance innovation and familiarity, what syntax is more readable and to whom.

就个人而言，我认为世界只需要这么多的FORTRAN 77实现。在某个时候，您会发现自己正在设计一种新的语言。 一旦开始这样做，方程式中较柔和，人性化的一面就变得至关重要。 诸如哪些功能易于学习，如何在创新和熟悉度之间取得平衡，哪种语法更易读以及对谁有帮助[6]。

> 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.

所有这些都会对您的新语言的成功产生深远的影响。 我希望您的语言取得成功，因此在某些章节中，我以一篇“设计笔记”结尾，这些是关于编程语言的人文方面的一些文章。我并不是这方面的专家——我不确定是否有人真的精通这些，因此，请您在阅读这些文字的时候仔细评估。这样的话，这些文字就能成为您思考的食材，这也正是我的目标。

> ## 1.3 The First Interpreter

1.3 第一个解释器

> 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.

我们将用Java编写第一个解释器jlox。（这里的）主要关注点是概念。 我们将编写最简单，最干净的代码，以正确实现该语言的语义。 这样能够帮助我们熟悉基本技术，并磨练对语言表现形式的确切理解。

> 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 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.

Java是一门很适合这种场景的语言。它的级别足够高，我们不会被繁琐的实现细节淹没，但代码仍是非常明确的。与脚本语言不同的是，它的底层没有隐藏太过复杂的机制，你可以使用静态类型来查看正在处理的数据结构。

> 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.

我选择Java还有特别的原因，就是因为它是一种面向对象的语言。 这种范式在90年代席卷了整个编程世界，如今已成为数百万程序员的主流思维方式。 很有可能您已经习惯了将代码组织到类和方法中，因此我们将让您在舒适的环境中学习。

> 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.

虽然学术语言专家有时瞧不起面向对象语言，但事实上，它们即使在语言工作中也被广泛使用。GCC和LLVM是用c++编写的，大多数JavaScript虚拟机也是这样。 面向对象的语言无处不在，并且针对该语言的工具和编译器通常是用同一种语言编写的[7]。

> 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.

最后，Java非常流行。 这意味着您很有可能已经了解它了，所以你要学习的东西就更少了。 如果您不太熟悉Java，也请不要担心。 我尽量只使用它的最小子集。我使用Java 7中的菱形运算符使代码看起来更简洁，但就“高级”功能而言，仅此而已。 如果您了解其它面向对象的语言（例如C＃或C ++），就没有问题。

> By the end of part II, we'll have a simple, readable implementation. What we won't have is a *fast* one. It also takes advantage of the Java virtual machine's own runtime facilities. We want to learn how Java *itself* implements those things.

在第二部分结束时，我们将得到一个简单易读的实现。 但是我们得到的不会是一个快速的解释器。它还是利用了Java虚拟机自身的运行时工具。我们想要学习Java本身是如何实现这些东西的。

> ## #1.4 The Second Interpreter

#1.4 第二个解释器

> 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.

所以在下一部分，我们将从头开始，但这一次是用C语言。C语言是理解实现编译器工作方式的完美语言，一直到内存中的字节和流经CPU的代码。

> 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.

我们使用C语言的一个重要原因是，我可以向您展示C语言特别擅长的东西，但这并不意味着您需要非常熟练地使用它。您不必是丹尼斯·里奇（Dennis Ritchie）的转世，但也不应被指针吓倒。

> 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.

如果你（对C的掌握）还没到那一步，找一本关于C的入门书，仔细阅读，读完后再回来。作为回报，从这本书中你将成为一个更优秀的C程序员。可以想想有多少语言实现是用C完成的: Lua、CPython和Ruby s MRI等，这里仅举几例。

> 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 it.

在我们的C解释器clox中[8]，我们不得不自己实现那些Java免费提供给我们的东西。 我们将编写自己的动态数组和哈希表。 我们将决定对象在内存中的表示方式，并构建一个垃圾回收器来回收它。

> 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 com-

> piler 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.

我们的Java版实现专注于正确性。 既然我们已经完成了，那么我们就要变得越来越快。 我们的C解释器将包含一个编译器[9]，该编译器会将Lox转换为有效的字节码形式（不用担心，我很快就会讲解这是什么意思）之后它会执行对应的字节码。 这与Lua，Python，Ruby，PHP和许多其它成功语言的实现所使用的技术相同。

> 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.

我们甚至会尝试进行基准测试和优化。 到最后，我们将为lox语言提供一个强大，准确，快速的解释器，并能够不落后于其他专业水平的实现。对于一本书和几千行代码来说已经不错了。

#CHALLENGES

#习题

> 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?

在我编写的这个小系统中，至少有六种特定领域语言（DSL），它们是什么?

> 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.

使用Java编写并运行一个“Hello, world!”程序，设置你需要的makefile或IDE项目使其正常工作。如果您有调试器，请先熟悉一下，并在程序运行时对代码逐步调试。

> 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.

对C也进行同样的操作。为了练习使用指针，可以定义一个堆分配字符串的双向链表。编写函数以插入，查找和删除其中的项目。 测试编写的函数。

> ## DESIGN NOTE: WHAT’S IN A NAME?

设计笔记：名称是什么?

> 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 sylla-

> bles 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 guard for, but it's worth considering. The designer of Nimrod ended up renaming his language to "Nim" because too many people only remember that Bugs Bunny used "Nimrod" (ironically, actually) as an insult.
>
> 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.

写这本书最困难的挑战之一是为它所实现的语言取个名字。我翻了好几页的备选名才找到一个合适的。当你某一天开始构建自己的语言时，你就会发现命名是非常困难的。一个好名字要满足几个标准：

1. **尚未使用。**如果您不小心使用了别人的名字，就可能会遇到各种法律和社会上的麻烦。
2. **容易发音。**如果一切顺利，将会有很多人会说和写您的语言名称。 超过几个音节或几个字母的任何内容都会使他们陷入无休止的烦恼。
3. **足够独特，易于搜索。**人们会Google你的语言的名字来了解它，所以你需要一个足够独特的单词，以便大多数搜索结果都会指向你的文档。不过，随着人工智能搜索引擎数量的增加，这已经不是什么大问题了。但是，如果您将语言命名为“ for”，那对用户基本不会有任何帮助。
4. **在多种文化中，都没有负面的含义。**这很难防范，但是值得深思。Nimrod的设计师最终将其语言重命名为“ Nim”，因为太多的人只记得Bugs Bunny使用“

Nimrod”作为一种侮辱（其实是讽刺）。

如果你潜在的名字通过了考验，就保留它吧。不要纠结于寻找一个能够抓住你语言精髓的名称。如果说世界上其他成功的语言的名字教会了我们什么的话，那就是名字并不重要。您所需要的只是一个相当独特的标记。

---

[^1]
静态类型系统尤其需要严格的形式推理。破解类型系统就像证明数学定理一样。事实证明这并非巧合。 上世纪初，Haskell Curry和William Alvin Howard证明了它们是同一枚硬币的两个方面：Curry-Howard同构。

[^2]
pidgins，洋泾浜语言，一种混杂的英语

[^3]
Yacc是一个工具，它接收语法文件并生成编译器的源文件，因此它有点像一个输出“编译器”的编译器，在术语中叫作“compiler-compiler”，即编译器的编译器。Yacc并不是同类工具中的第一个，这就是为什么它被命名为“Yacc”—*Yet Another* Compiler-Compiler(另一个Compiler-Compiler)。后来还有一个类似的工具是Bison，它的名字源于Yacc和yak的发音，是一个双关语。

[^4]
警告:挑战题目通常要求您对正在构建的解释器进行更改。您需要在代码副本中实现这些功能。后面的章节都假设你的解释器处于原始(未解决挑战题)状态。

[^5]
我知道很多语言黑客的职业就基于此。您将一份语言规范塞到他们的门下，等上几个月，代码和基准测试结果就出来了。

[^6]
希望您的新语言不会将对打孔卡宽度的假设硬编码到语法中。

[^7]
编译器以一种语言读取文件。 翻译它们，并以另一种语言输出文件。 您可以使用任何一种语言（包括与目标语言相同的语言）来实现编译器，该过程

称为“自举”。你现在还不能使用编译器本身来编译你自己的编译器，但是如果你用其它语言为你的语言写了一个编译器，你就可以用那个编译器编译一次你的编译器。现在，您可以使用自己的编译器的已编译版本来编译自身的未来版本，并且可以从另一个编译器中丢弃最初的已编译版本。 这就是所谓的“自举”，通过自己的引导程序将自己拉起来。

[^8]

我把这个名字读作“sea-locks”，但是你也可以读作“clocks”，如果你愿意的话可以像希腊人读“x”那样将其读作“clochs”，

[^9]

你以为这只是一本讲解释器的书吗?它也是一本讲编译器的书。买一送一。