目录

第一部分

欢迎大家

这可能将会是一个伟大旅程的开始。编程语言这个领域有着巨大的探索和玩耍的空间。你可以在这个大房子里尽情的创造，可以把你做的东西分享给别人，也可以仅仅是娱乐自己。很多伟大的计算机科学家和软件工程师将他们一生的精力都投入到了这个领域，而这个领域却还远远没有到头。如果这本书是你踏入这个领域所接触的第一本书，欢迎你！

这本书将带着你在编程语言的世界里旅行一番。但在我们穿好登山靴出去旅行之前，我们需要先熟悉一下我们要旅行的地点的整个地图。这一部分的章节将会带我们学习一些编程语言中用到的基本概念，以及这些概念的组织方式。

我们也会对 Lox 这门语言非常熟悉。因为我们将要在书中剩下的部分实现这门语言（要实现两遍！）。

第一章 简介

> 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。
>
> ——盖曼

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[1]。

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。

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。

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。

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 为什么要学习这些东西？

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。

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]这里要和我的家人和朋友们说声抱歉，抱歉这些年我是如此的不着调，如此的心不在焉（脑子里一直在写书）！

1.1.1 小型编程语言无处不在

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，如应用程序脚本语言、模板引擎、标记格式和配置文件。

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。

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 自己实现编程语言是一种很好的锻炼

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。

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。

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 一个额外的原因

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。

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。

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。

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。

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。

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 本书的组织方式

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。

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。

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。

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 代码

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。

许多其他的语言书籍和语言实现都使用 Lex 和 Yacc 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。

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。

一本书和“真实世界”的条件是有区别的，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。可能我对某些写法是无所谓的，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。

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//。

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 代码片段

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。

一个完整的代码片段可能如下所示：

lox/Scanner.java，在 scanToken() 中替换 1 行

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

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 题外话

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，将旁白信息作为脚注附在章节之后】

1.2.4 挑战

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。

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。

1.2.5 设计笔记

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。

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。

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 第一个解释器

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 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。

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。

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。

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。

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 第二个解释器

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。

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。

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 的 MRI 等，这里仅举几例。

在我们的 C 解释器 clox 中，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。

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，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。

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。

1.5 挑战

1. 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. 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. 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。

1.6 设计笔记：名字是什么？

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。

第二章 全书地图

> 你必须要有一张地图，无论它是多么粗糙。否则你就会到处乱逛。在《指环王》中，我从未让任何人在某一天走得超出他力所能及的范围。
>
> ——托尔金

我们不想到处乱逛，所以在我们开始之前，让我们先浏览一下以前的语言实现者所绘制的领土。它能帮助我们了解我们的目的地和其他人采用的备选路线。

首先，我先做个简单说明。本书的大部分内容都是关于语言的实现，它与语言本身这种柏拉图式的理想形式有所不同。诸如“堆栈”，“字节码”和“递归下降”之类的东西是某个特定实现中可能使用的基本要素。从用户的角度来说，只要最终产生的装置能够忠实地遵循语言规范，它内部的都是实现细节。

我们将会花很多时间在这些细节上，所以如果我每次提及的时候都写“语言实现”，我的手指都会被磨掉。相反，除非有重要的区别，否则我将使用“语言”来指代一种语言或该语言的一种实现，或两者皆有。

2.1 语言的各部分

自计算机的黑暗时代以来，工程师们就一直在构建编程语言。当我们可以和计算机对话的时候，我们发现这样做太难了，于是我们寻求电脑的帮助。我觉得很有趣的是，即使今天的机器确实快了一百万倍，存储空间也大了几个数量级，但我们构建编程语言的方式几乎没有改变。

尽管语言设计师所探索的领域辽阔，但他们所走过的路却很少。并非每种语言都采用完全相同的路径（有些采用一种或两种捷径），但除此之外，从海军少将 Grace Hopper 的第一个 COBOL 编译器，一直到一些热门的新移植到 JavaScript 的语言，它们的“文档”完全是由 Git 仓库中一个编辑得很差的 README 组成的。

我把一个语言实现可能选择的路径网络类比为爬山。你从最底层开始，程序是原始的源文本，实际上只是一串字符。每个阶段都会对程序进行分析，并将其转换为更高层次的表现形式，从而使语义（作者希望计算机做什么）变得更加明显。

最终我们达到了峰顶。我们可以鸟瞰用户的程序，可以看到他们的代码含义是什么。我们开始从山的另一边下山。我们将这个最高级的表示形式转化为连续的较低级别的形式，从而越来越接近我们所知道的如何让 CPU 真正执行的形式。

让我们沿着每一条路线和每一个感兴趣的地方走一遍。我们的旅程从左边的用户源代码的纯文本开始。

2.1.1 扫描

第一步是**扫描**，也就是所谓的**词法**，或者说（如果你想给别人留下深刻印象）**词法分析**。它们的意思都差不多。我喜欢“lexing”，因为这听起来像是一个邪恶的超级大坏蛋会做的事情，但我还是用“scanning”，因为它似乎更常见一些。

扫描器 (或词法解析器) 接收线性字符流，并将它们组合成一系列更类似于“单词”的东西。在编程语言中，这些词的每一个都被称为**标记**。有些标记是单个字符，比如 (和,。其他的可能是几个字符长的，比如数字（123）、字符串字元（"hi!"）和标识符（min）。

源文件中的一些字符实际上没有任何意义。空格通常是无关紧要的，而注释，从定义就能看出来，会被语言忽略。扫描仪通常会丢弃这些字符，留下一个干净的有意义的标记序列。

2.1.2 语法分析

下一步是**解析**。这就是我们从句法中得到**语法**的地方——语法能够将较小的部分组成较大的表达式和语句。你在英语课上画过句子图吗？如果有，你就做了解析器所做的事情，区别在于，英语中有成千上万的“关键字”和大量的歧义，而编程语言要简单得多。

解析器接受标记的平面序列，并构建反映语法嵌套本质的树结构。这些树有两个不同的名称：**解析树**或**抽象语法树**，这取决于它们与源语言的语法结构有多接近。在实践中，语言黑客通常称它们为“**语法树**”、“**AST**”，或者干脆直接说“**树**”。

解析在计算机科学中有着悠久而丰富的历史，它与人工智能界有着密切的联系。今天用于解析编程语言的许多技术最初是由人工智能研究人员设想的，他们试图让计算机与我们对话，以解析人类语言。

事实证明，相对那些解析器能够处理的严格语法来说，人类语言太混乱了；但对于编程语言中更简单的人工语法来说，人类语言却是完美的。唉，可惜我们这些有缺陷的人类仍然会错误地使用这些简单的语法，因此解析器的工作还包括通过报告**语法错误**让我们知道出错了。

2.1.3 静态分析

在所有实现中，前两个阶段都非常相似。现在，每种语言的个性化特征开始发挥作用。至此，我们知道了代码的语法结构（诸如哪些表达式嵌套在其他表达式中）之类的东西，但是我们知道的也就仅限于此了。

在 a+b 这样的表达式中，我们知道我们要把 a 和 b 相加，但我们不知道这些名字指的是什么。它们是局部变量吗？全局变量？它们在哪里被定义？

大多数语言所做的第一点分析叫做**绑定**或**解析**。对于每一个**标识符**，我们都要找出定义该名称的地方，并将两者连接起来。这就是**作用域**的作用—在这个源代码区域中，某个名字可以用来引用某个声明。

如果语言是静态类型的，这时我们就进行类型检查。一旦我们知道了 a 和 b 的声明位置，我们也可以弄清楚它们的类型。然后如果这些类型不支持互相累加，我们就会报告一个**类型错误**。

深吸一口气。我们已经到达了山顶，并对用户的程序有了全面的了解。所有这些从分析中可见的语义信息都需要存储在某个地方。我们可以把它藏在几个地方：

- 通常，它会被直接存储在语法树本身的**属性**中—属性是节点中的额外字段，这些字段在解析时不会初始化，但在稍后会进行填充。
- 有时，我们可能会将数据存储在外部的查找表中。通常，该表的关键字是标识符，即变量和声明的名称。在这种情况下，我们称其为**符号表**，并且其中与每个键关联的值告诉我们该标识符所指的是什么。
- 最强大的记录工具是将树转化为一个全新的数据结构，更直接地表达代码的语义。这是下一节的内容。

到目前为止，所有内容都被视为实现的**前端**。您可能会猜至此以后是**后端**，其实并不是。在过去的年代，当“前端”和“后端”被创造出来时，编译器要简单得多。后来，研究人员在两个半部之间引入了新阶段。威廉 · 沃尔夫（William Wulf）和他的同伴没有放弃旧术语，而是新添加了一个迷人但有点自相矛盾的名称“**中端**”。

2.1.4 中间表示

你可以把编译器看成是一条流水线，每个阶段的工作是把代表用户代码的数据组织起来，使下一阶段的实现更加简单。管道的前端是针对程序所使用的源语言编写的。后端关注的是程序运行的最终架构。

在中间阶段，代码可能被存储在一些**中间表示**（**intermediate representation**，也叫**IR**）中，这些中间表示与源文件或目标文件形式都没有紧密的联系（因此叫作“中间”）。相反，IR 充当了这两种语言之间的接口。

这可以让你更轻松地支持多种源语言和目标平台。假设你想实现 Pascal、C 和 Fortran 编译器，并且你的目标平台的体系结构是：x86、ARM，还有 SPARC。通常情况下，这意味着你需要写九个完整的编译器：Pascal → x86，C → ARM，以及其他各种组合。

一个共享的中间表示可以大大减少这种情况。你为每个产生 IR 的源语言写一个前端。然后为每个目标平台写一个后端。现在，你可以将这些混搭起来，得到每一种组合。

还有一个重要的原因是，我们可能希望将代码转化为某种形式，使语义更加明确.....。

2.1.5 优化

一旦我们理解了用户程序的含义，我们就可以自由地用另一个具有相同语义但实现效率更高的程序来交换它—我们可以对它进行**优化**。

一个简单的例子是**常量折叠**：如果某个表达式求值得到的始终是完全相同的值，我们可以在编译时进行求值，并用其结果替换该表达式的代码。如果用户输入：

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

我们可以在编译器中完成所有的算术运算，并将代码更改为：

```
pennyArea = 0.4417860938;
```

优化是编程语言业务的重要组成部分。许多语言黑客把他们的整个职业生涯都花在了这里，竭尽所能地从他们的编译器中挤出每一点性能，以使他们的基准测试速度提高一个百分点。这可能会成为一种困扰。

我们通常会跳过本书中的棘手问题。许多成功的语言令人惊讶地很少进行编译期优化。例如，Lua 和 CPython 生成相对未优化的代码，并将其大部分性能优化工作集中在运行时上。

2.1.6 代码生成

我们已经将所有可以想到的优化应用到了用户程序中。最后一步是将其转换为机器可以实际运行的形式。换句话说，**生成代码**（或**代码生成**），这里的“代码”通常是指 CPU 运行的类似于汇编的原始指令，而不是人类可能想要阅读的“源代码”。

最后，我们到了**后端**，从山的另一侧开始向下。从现在开始，随着我们越来越接近于思维简单的机器可以理解的东西，我们对代码的表示变得越来越原始，就像逆向进化。

我们需要做一个决定。我们是为真实 CPU 还是虚拟 CPU 生成指令？如果我们生成真实的机器代码，则会得到一个可执行文件，操作系统可以将其直接加载到芯片上。原生代码快如闪电，但生成它需要大量工作。当今的体系结构包含大量指令，复杂的流水线和足够塞满一架 747 行李舱的历史包袱。

使用芯片的语言也意味着你的编译器是与特定的架构相绑定的。如果你的编译器以 x86 机器代码为目标，那么它就无法在 ARM 设备上运行。一直到 60 年代，在计算机体系结构的寒武纪大爆发期间，这种缺乏可移植性的情况是一个真正的障碍。

为了解决这个问题，像 BCPL 的 Martin Richards 和 Pascal 的 Niklaus Wirth 这样的黑客，让他们的编译器生成虚拟机代码。他们不是为真正的芯片编写指令，而是为一个假设的、理想化的机器编写代码。Wirth 称这种 **p-code** 为“可移植代码”，但今天，我们通常称它为**字节码**，因为每条指令通常都是一个字节长。

这些合成指令的设计是为了更紧密地映射到语言的语义上，而不必与任何一个计算机体系结构的特性和它积累的历史错误绑定在一起。你可以把它想象成语言底层操作的密集二进制编码。

2.1.7 虚拟机

如果你的编译器产生了字节码，你的工作还没有结束。因为没有芯片可以解析这些字节码，因此你还需要进行翻译。同样，您有两个选择。您可以为每个目标体系结构编写一个小型编译器，将字节码转换为该机器的本机代码。您仍然需要针对您支持的每个芯片做一些工作，但最后这个阶段非常简单，你可以在你支持的所有机器上重复使用编译器管道的其余部分。你基本上是把你的字节码作为一个中介码。

或者，您可以编写**虚拟机（VM）**，该程序可在运行时模拟支持虚拟架构的虚拟芯片。在虚拟机中运行字节码比提前将其翻译成本地代码要慢，因为每条指令每次执行时都必须在运行时模拟。作为回报，你得到的是简单性和可移植性。用比如说 C 语言实现你的虚拟机，你就可以在任何有 C 编译器的平台上运行你的语言。这就是我们在本书中构建的第二个解释器的工作原理。

2.1.8 运行时

我们终于将用户程序锤炼成可以执行的形式。最后一步是运行它。如果我们将其编译为机器码，我们只需告诉操作系统加载可执行文件，然后就可以运行了。如果我们将它编译成字节码，我们需要启动 VM 并将程序加载到其中。

在这两种情况下，除了最基本的底层语言外，我们通常需要我们的语言在程序运行时提供一些服务。例如，如果语言自动管理内存，我们需要一个垃圾收集器去回收未使用的比特位。如果我们的语言支持 instance of 测试，这样你就可以看到你有什么类型的对象，那么我们就需要一些表示方法来跟踪执行过程中每个对象的类型。

所有这些东西都是在运行时进行的，所以它被恰当地称为，**运行时**。在一个完全编译的语言中，实现运行时的代码会直接插入到生成的可执行文件中。比如说，在 Go 中，每个编译后的应用程序都有自己的一份 Go 的运行时副本直接嵌入其中。如果语言是在解释器或虚拟机内运行，那么运行时将驻留于虚拟机中。这也就是 Java、Python 和 JavaScript 等大多数语言实现的工作方式。

2.2 捷径和备选路线

这是一条漫长的道路，涵盖了你要实现的每个可能的阶段。许多语言的确走完了整条路线，但也有一些捷径和备选路径。

2.2.1 单遍编译器

一些简单的编译器将解析、分析和代码生成交织在一起，这样它们就可以直接在解析器中生成输出代码，而无需分配任何语法树或其他 IR。这些**单遍编译器**限制了语言的设计。您没有中间数据结构来存储程序的全局信息，也不会重新访问任何之前解析过的代码部分。这意味着，一旦您看到某个表达式，就需要足够的知识来正确地对其进行编译。

Pascal 和 C 语言就是围绕这个限制而设计的。在当时，内存非常珍贵，一个编译器可能连整个源文件都无法存放在内存中，更不用说整个程序了。这也是为什么 Pascal 的语法要求类型声明要先出现在一个块中。这也是为什么在 C 语言中，你不能在定义函数的代码上面调用函数，除非你有一个明确的前向声明，告诉编译器它需要知道什么，以便生成调用后面函数的代码。

2.2.2 树遍历解释器

有些编程语言在将代码解析为 AST 后就开始执行代码（可能应用了一点静态分析）。为了运行程序，解释器每次都会遍历语法树的一个分支和叶子，并在运行过程中计算每个节点。

这种实现风格在学生项目和小型语言中很常见，但在通用语言中并不广泛使用，因为它往往很慢。有些人使用“解释器”仅指这类实现，但其他人对“解释器”一词的定义更宽泛，因此我将使用没有歧义的“**树遍历解释器**”来指代这些实现。我们的第一个解释器就是这样工作的。

2.2.3 转译器

为一种语言编写一个完整的后端可能需要大量的工作。如果您有一些现有的通用 IR 作为目标，则可以将前端转换到该 IR 上。否则，您可能会陷入困境。但是，如果您将某些其他*源语言*视为中间表示，该怎么办？

您需要为您的语言编写一个前端。然后，在后端，您可以生成一份与您的语言级别差不多的其他语言的有效源代码字符串，而不是将所有代码降低到某个原始目标语言的语义。然后，您可以使用该语言现有的编译工具作为逃离大山的路径，得到某些可执行的内容。

人们过去称之为**源到源编译器**或**转换编译器**。随着那些为了在浏览器中运行而编译成 JavaScript 的各类语言的兴起，它们有了一个时髦的名字—**转译器**。

虽然第一个编译器是将一种汇编语言翻译成另一种汇编语言，但现今，大多数编译器都适用于高级语言。在 UNIX 病毒式地传播到各种各样的机器之后，开始了一个悠久的编译器传统，即编译器以 C 作为输出语言。只要 UNIX 存在，就可以使用 C 编译器，并生成有效的代码，因此，以 C 为目标是让语言在许多体系结构上运行的好方法。

Web 浏览器是今天的“机器”，它们的“机器代码”是 JavaScript，所以现在似乎几乎所有的语言都有一个以 JS 为目标的编译器，因为这是让你的代码在浏览器中运行的主要方式。

编译器的前端（扫描器和解析器）看起来跟其他编译器相似。然后，如果源语言只是在目标语言之上包装的简单语法外壳，则它可能会完全跳过分析，并直接输出目标语言中的类似语法。

如果两种语言的语义差异较大，那么你就会看到完整编译器的更多典型阶段，包括分析甚至优化。然后，在代码生成阶段，无需输出一些像机器代码一样的二进制语言，而是在目标语言中生成一串语法正确的源码（好吧，目标代码）。

不管是哪种方式，你再通过目标语言已有的编译管道运行生成的代码，就可以了。

2.2.4 即时编译

最后一个与其说是捷径，不如说是危险的高山争霸赛，最好留给专家。执行代码最快的方法是将代码编译成机器代码，但你可能不知道你的最终用户的机器支持什么架构。

该怎么做呢?

您可以做和 HotSpot JVM、Microsoft 的 CLR 和大多数 JavaScript 解释器相同的事情。在终端用户的机器上，当程序加载时（无论是从 JS 中还是从源代码加载，或者是 JVM 和 CLR 的平台无关的字节码），都可以将其编译为对应的本地代码，以适应本机支持的体系结构。自然地，这被称为**即时编译**。大多数黑客只是说 “JIT”，其发音与 “fit” 押韵。

最复杂的 JIT 将性能分析钩子插入到生成的代码中，以查看哪些区域对性能最为关键，以及哪些类型的数据正在流经其中。然后，随着时间的推移，它们将通过更高级的优化功能自动重新编译那些热点部分。

2.3 编译器和解释器

现在我已经向你的脑袋里塞满了一大堆编程语言术语，我们终于可以解决一个自远古以来一直困扰着程序员的问题: 编译器和解释器之间有什么区别?

事实证明，这就像问水果和蔬菜的区别一样。这看上去似乎是一个非此即彼的选择，但实际上 “水果” 是一个植物学术语，“蔬菜” 是烹饪学术语。严格来说，一个并不意味着对另一个的否定。有不是蔬菜的水果（苹果），也有不是水果的蔬菜（胡萝卜），也有既是水果又是蔬菜的可食用植物，比如西红柿。

好，回到语言上:

- **编译**是一种实现技术，其中涉及到将源语言翻译成其他语言-通常是较低级的形式。当你生成字节码或机器代码时，你就是在编译。当你移植到另一种高级语言时，你也在编译。
- 当我们说语言实现 “是**编译器**” 时，是指它会将源代码转换为其他形式，但不会执行。用户必须获取结果输出并自己运行。
- 相反，当我们说一个实现 “是一个**解释器**” 时，是指它接受源代码并立即执行它。它 “从源代码” 运行程序。

像苹果和橘子一样，某些实现显然是编译器，而不是解释器。GCC 和 Clang 接受您的 C 代码并将其编译为机器代码。最终用户直接运行该可执行文件，甚至可能永远都不知道使用了哪个工具来编译它。所以这些是 C 的*编译器*。

在 Matz 的旧版本的 Ruby 规范实现中，用户从源代码中运行 Ruby。该实现通过遍历语法树对其进行解析并直接执行。期间都没有发生其他的转换，无论是在实现内部还是以任何用户可见的形式。所以这绝对是一个 Ruby 的*解释器*。

但是 CPython 呢? 当你使用它运行你的 Python 程序时，代码会被解析并转换为内部字节码格式，然后在虚拟机内部执行。从用户的角度来看，这显然是一个解释器——他们是从源代码开始运行自己的程序。但如果你看一下 CPython 的内部，你会发现肯定有一些编译工作在进行。

答案是两者兼而有之。CPython 是一个解释器，但他也有一个编译器。实际上，大多数脚本语言都以这种方式工作，如您所见:

中间那个重叠的区域也是我们第二个解释器所在的位置，因为它会在内部编译成字

节码。所以，虽然本书名义上是关于解释器的，但我们也会涉及一些编译的内容。

2.4 我们的旅程

一下子有太多东西要消化掉。别担心。这一章并不是要求你理解所有这些零碎的内容。我只是想让你们知道它们是存在的，以及大致了解它们是如何组合在一起的。

当您探索本书本书所指导的路径之外的领域时，这张地图应该对你很有用。我希望你自己出击，在那座山里到处游走。

但是，现在，是我们自己的旅程开始的时候了。系好你的鞋带，背好你的包，走吧。从这里开始，你需要关注的是你面前的路。

2.5 挑战

1. 选择一个你喜欢的语言的开源实现。下载源代码，并在其中探索。试着找到实现扫描器和解析器的代码，它们是手写的，还是用 Lex 和 Yacc 等工具生成的？（存在.l 或.y 文件通常意味着后者）
2. 实时编译往往是实现动态类型语言最快的方法，但并不是所有的语言都使用它。有什么理由不采用 JIT 呢？
3. 大多数可编译为 C 的 Lisp 实现也包含一个解释器，该解释器还使它们能够即时执行 Lisp 代码。为什么？

第 三 章 Lox 编程语言

> 你能够为别人做什么比做早餐更好的事情吗?
>
> ——布尔丹

我们将用本书的其余部分来照亮 Lox 语言的每一个黑暗和杂乱的角落，但如果让你在对目标一无所知的情况下，就立即开始为解释器编写代码，这似乎很残忍。

与此同时，我也不想在您编码之前，就把您拖入大量的语言和规范术语中。所以这是一个温和、友好的 Lox 介绍，它会省去很多细节和边缘情况。后面我们有足够的时间来解决这些问题。

3.1 Hello, Lox

下面是你对 Lox 的第一次体验：

```
// Your first Lox program!
print "Hello, world!";
```

正如那句//行注释和后面的分号所暗示的那样，Lox 的语法是 C 语言家族的成员之一。(因为 print 是一个内置语句，而不是库函数，所以字符串周围没有括号。)

这里，我并不是想说 C 语言具有出色的语法。如果我们想要一些优雅的东西，我们可能会模仿 Pascal 或 Smalltalk。如果我们想要完全体现斯堪的纳维亚家具的极简主义风格，我们会实现一个 Scheme。这些都有其优点。

但是，类 C 的语法所具有的反而是一些在语言中更有价值的东西：熟悉度。我知道你已经对这种风格很熟悉了，因为我们将用来实现 Lox 的两种语言——Java 和 C——也继承了这种风格。让 Lox 使用类似的语法，你就少了一件需要学习的事情。

3.2 高级语言

虽然这本书最终比我所希望的要大，但它仍然不够大，无法将 Java 这样一门庞大的语言放进去。为了在有限的篇幅里容纳两个完整的 Lox 实现，Lox 本身必须相当紧凑。

当我想到那些小而有用的语言时，我脑海中浮现的是像 JavaScript、Scheme 和 Lua 这样的高级“脚本”语言。在这三种语言中，Lox 看起来最像 JavaScript，主要是因为大多数 C 语法语言都是这样的。稍后我们将了解到，Lox 实现作用域的方式与 Scheme 密切相关。我们将在第三部分中构建的 C 风格的 Lox 很大程度上得益于 Lua 的干净、高效的实现。

Lox 与这三种语言有两个共同之处：

3.2.1 动态类型

Lox 是动态类型的。变量可以存储任何类型的值，单个变量甚至可以在不同时间存储不同类型的值。如果尝试对错误类型的值执行操作（例如，将数字除以字符串），则会在运行时检测到错误并报告。

喜欢静态类型的原因有很多，但它们都比不上为 Lox 选择动态类型的实际原因。静态类型系统需要学习和实现大量的工作。跳过它会让你的语言更简单，也可以让本书更短。如果我们将类型检查推迟到运行时，我们将可以更快地启动解释器并执行代码。

3.2.2 自动内存管理

高级语言的存在是为了消除容易出错的低级工作，还有什么比手动管理存储的分配和释放更繁琐的呢？没有人会抬起头来迎接早晨的阳光，“我迫不及待想找到正确的位置，调用 free() 方法来准备今天每个字节需要分配的内存！”

有两种主要的内存管理技术：**引用计数**和**跟踪垃圾收集**（通常仅称为“**垃圾收集**”或“**GC**”）。引用计数器的实现要简单得多——我想这就是为什么 Perl、PHP 和 Python 一开始都使用该方式的原因。但是，随着时间的流逝，引用计数的限制变得太麻烦了。所有这些语言最终都添加了完整的跟踪 GC 或至少一种足以清除对象循环的管理方式。

追踪垃圾收集有一个可怕的名声。在原始内存的层面上工作是有点折磨人的。调试 GC 有时会让你在梦中看到 hex dumps。但是，请记住，这本书是关于驱散魔法和杀死那些怪物的，所以我们要写出自己的垃圾收集器。我想你会发现这个算法相当简单，而且实现起来很有趣。

3.3 数据类型

在 Lox 的小宇宙中，构成所有物质的原子是内置的数据类型。只有几个：

Booleans——没有逻辑就不能编码，没有布尔值也就没有逻辑。“真”和“假”，就是软件的阴与阳。与某些古老的语言重新利用已有类型来表示真假不同，Lox 具有专用的布尔类型。在这次探险中，我们可能会有些粗暴，但我们不是野蛮人。

显然，有两个布尔值，每个值都有一个字面量：

```
true; // Not false.
false; // Not *not* false.
```

Numbers——Lox 只有一种数字：双精度浮点数。由于浮点数还可以表示各种各样的整数，因此可以覆盖很多领域，同时保持简单。

功能齐全的语言具有多种数字语法——十六进制，科学计数法，八进制和各种有趣的东西。我们只使用基本的整数和十进制文字：

```
1234; // An integer.
12.34; // A decimal number.
```

Strings——在第一个示例中，我们已经看到一个字符串字面量。与大多数语言一样，它们用双引号引起来：

```
"I am a string";
"";   // The empty string.
"123"; // This is a string, not a number.
```

我们在实现它们时会看到，在这个无害的字符序列中隐藏了相当多的复杂性。

Nil——还有最后一个内置数据，它从未被邀请参加聚会，但似乎总是会出现。它代表“没有价值”。在许多其他语言中称为“null”。在 Lox 中，我们将其拼写为 nil。（当我们实现它时，这将有助于区分 Lox 的 nil 与 Java 或 C 的 null）

有一些很好的理由表明在语言中不使用空值是合理的，因为空指针错误是我们行业的祸害。如果我们使用的是静态类型语言，那么禁止它是值得的。然而，在动态类型中，消除它往往比保留它更加麻烦。

3.4 表达式

如果内置数据类型及其字面量是原子，那么表达式必须是分子。其中大部分大家都很熟悉。

3.4.1 算术运算

Lox 具备了您从 C 和其他语言中了解到的基本算术运算符：

```
add + me;
subtract - me;
multiply * me;
divide / me;
```

操作符两边的子表达式都是**操作数**。因为有两个操作数，它们被称为二**元**运算符 (这与二进制的 1 和 0 二元没有关联)。由于操作符固定在操作数的中间，因此也称为**中缀**操作符，相对的，还有**前缀**操作符 (操作符在操作数前面) 和**后缀**操作符 (操作符在操作数后面)。

有一个数学运算符既是中缀运算符也是前缀运算符，-运算符可以对数字取负：

```
-negateMe;
```

所有这些操作符都是针对数字的，将任何其他类型操作数传递给它们都是错误的。唯一的例外是 + 运算符——你也可以传给它两个字符串将它们串接起来。

3.4.2 比较与相等

接下来，我们有几个返回布尔值的操作符。我们可以使用旧的比较操作符来比较数字 (并且只能比较数字)：

```
less < than;
lessThan <= orEqual;
greater > than;
greaterThan >= orEqual;
```

我们可以测试两个任意类型的值是否相等：

```
1 == 2;       // false.
"cat" != "dog"; // true.
```

即使是不同类型也可以：

```
314 == "pi"; // false.
```

不同类型的值 * 永远不会 * 相等：

```
123 == "123"; // false.
```

我通常是反对隐式转换的。

3.4.3 逻辑运算

取非操作符，是前缀操作符!，如果操作数是 true，则返回 false，反之亦然：

```
!true;  // false.
!false; // true.
```

其他两个逻辑操作符实际上是表达式伪装下的控制流结构。and 表达式用于确认两个操作数是否都是 true。如果左侧操作数是 false，则返回左侧操作数，否则返回右侧操作数：

```
true and false; // false.
true and true;  // true.
```

or 表达式用于确认两个操作数中任意一个（或者都是）为 true。如果左侧操作数为 true，则返回左侧操作数，否则返回右侧操作数：

```
false or false; // false.
true or false;  // true.
```

and 和 or 之所以像控制流结构，是因为它们会**短路**。如果左操作数为假，and 不仅会返回左操作数，在这种情况下，它甚至不会计算右操作数。反过来，（“相对的”？）如果 or 的左操作数为真，右操作数就会被跳过。

3.4.4 优先级与分组

所有这些操作符都具有与 c 语言相同的优先级和结合性 (当我们开始解析时，会进行更详细的说明)。在优先级不满足要求的情况下，你可以使用 () 来分组：

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

我把其他典型的操作符从我们的小语言中去掉了，因为它们在技术上不是很有趣。没有位运算、移位、取模或条件运算符。我不是在给你打分，但如果你通过自己的方式来完成支持这些运算的 Lox 实现，你会在我心中得到额外的加分。

这些都是表达式形式 (除了一些与我们将在后面介绍的特定特性相关的)，所以让我们继续。

3.5 语句

现在我们来看语句。表达式的主要作用是产生一个 * 值 *，语句的主要作用是产生一个 * 效果 *。由于根据定义，语句不求值，因此必须以某种方式改变世界（通常是修改某些状态，读取输入或产生输出）才能有用。

您已经看到了几种语句。第一个是：

```
print "Hello, world!";
```

print 语句计算单个表达式并将结果显示给用户。您还看到了一些语句，例如：

```
"some expression";
```

表达式后跟分号（;）可以将表达式提升为语句状态。这被称为 (很有想象力) **表达式语句**。

如果您想将一系列语句打包成一个语句，那么可以将它们打包在一个块中：

```
{
  print "One statement.";
  print "Two statements.";
}
```

块还会影响作用域，我们将在下一节中进行说明。

3.6 变量

你可以使用 var 语句声明变量。如果你省略了初始化操作，变量的值默认为 nil：

```
var imAVariable = "here is my value";
var iAmNil;
```

一旦声明完成，你自然就可以通过变量名对其进行访问和赋值：

```
var breakfast = "bagels";
print breakfast; // "bagels".
breakfast = "beignets";
print breakfast; // "beignets".
```

我不会在这里讨论变量作用域的规则，因为我们在后面的章节中将会花费大量的时间来详细讨论这些规则。在大多数情况下，它的工作方式与您期望的 C 或 Java 一样。

3.7 控制流

如果你不能跳过某些代码，或者不能多次执行某些代码，就很难写出有用的程序。这意味着控制流。除了我们已经介绍过的逻辑运算符之外，Lox 直接从 C 中借鉴了三种语句类型。

if 语句根据某些条件执行两条语句中的一条：

```
if (condition) {
  print "yes";
} else {
  print "no";
}
```

只要条件表达式的计算结果为 true，while 循环就会重复执行循环体：

```
var a = 1;
while (a < 10) {
  print a;
  a = a + 1;
}
```

最后，还有 for 循环：

```
for (var a = 1; a < 10; a = a + 1) {
  print a;
}
```

这个循环与之前的 while 循环做同样的事情。大多数现代语言也有某种 for-in 或 foreach 循环，用于显式迭代各种序列类型。在真正的语言中，这比我们在这里使用的粗糙的 C 风格 for 循环要好。Lox 只保持了它的基本功能。

3.8 函数

函数调用表达式与 C 语言中一样：

```
makeBreakfast(bacon, eggs, toast);
```

你也可以在不传递任何参数的情况下调用一个函数：

```
makeBreakfast();
```

与 Ruby 不同的是，在本例中括号是强制性的。如果你把它们去掉，就不会调用函数，只是指向该函数。

如果你不能定义自己的函数，一门语言就不能算有趣。在 Lox 里，你可以通过 fun 完成：

```
fun printSum(a, b) {
  print a + b;
}
```

现在是澄清一些术语的好时机。有些人把 parameter 和 argument 混为一谈，好像它们可以互换，而对许多人来说，它们确实可以互换。我们要花很多时间围绕语义学来对其进行分辨，所以让我们在这里把话说清楚：

- **argument** 是你在调用函数时传递给它的实际值。所以一个函数调用有一个 *argument* 列表。有时你会听到有人用**实际参数**指代这些参数。
- **parameter** 是一个变量，用于在函数的主体里面存放参数的值。因此，一个函数声明有一个 *parameter* 列表。也有人把这些称为**形式参数**或者干脆称为**形参**。

函数体总是一个块。在其中，您可以使用 return 语句返回一个值：

```
fun returnSum(a, b) {
  return a + b;
}
```

如果执行到达代码块的末尾而没有 return 语句，则会隐式返回 nil。

3.8.1 闭包

在 Lox 中，函数是一等公民，这意味着它们都是真实的值，你可以对这些值进行引用、存储在变量中、传递等等。下面的代码是有效的：

```
fun addPair(a, b) {
  return a + b;
}

fun identity(a) {
  return a;
}

print identity(addPair)(1, 2); // Prints "3".
```

由于函数声明是语句，所以可以在另一个函数中声明局部函数：

```
fun outerFunction() {
  fun localFunction() {
    print "I'm local!";
  }

  localFunction();
}
```

如果将局部函数、头等函数和块作用域组合在一起，就会遇到这种有趣的情况：

```
fun returnFunction() {
  var outside = "outside";

  fun inner() {
    print outside;
  }

  return inner;
}

var fn = returnFunction();
fn();
```

在这里，inner() 访问了在其函数体外的外部函数中声明的局部变量。这样可行吗？现在很多语言都从 Lisp 借鉴了这个特性，你应该也知道答案是肯定的。

要做到这一点，inner() 必须“保留”对它使用的任何周围变量的引用，这样即使在外层函数返回之后，这些变量仍然存在。我们把能做到这一点的函数称为**闭包**。现在，这个术语经常被用于任何头类函数，但是如果函数没有在任何变量上闭包，那就有点用词不当了。

可以想象，实现这些会增加一些复杂性，因为我们不能再假定变量作用域严格地像堆栈一样工作，在函数返回时局部变量就消失了。我们将度过一段有趣的时间来学习如何使这些工作，并有效地做到这一点。

3.9 类

因为 Lox 具有动态类型、词法 (粗略地说，就是块) 作用域和闭包，所以它离函数式语言只有一半的距离。但正如您将看到的，它离成为一种面向对象的语言也有一半的距离。这两种模式都有很多优点，所以我认为有必要分别介绍一下。

类因为没有达到其宣传效果而受到抨击，所以让我先解释一下为什么我把它们放到 Lox 和这本书中。这里实际上有两个问题：

3.9.1 为什么任何语言都想要面向对象？

现在像 Java 这样的面向对象的语言已经销声匿迹了，只能在舞台上表演，喜欢它们已经不酷了。为什么有人要用对象来做一门新的语言呢？这不就像发行 8 轨音乐一样吗？

90 年代的“一直都是继承”的狂潮确实产生了一些畸形的类层次结构，但面向对象的编程还是很流行的。数十亿行成功的代码都是用 OOP 语言编写的，为用户提供了数百万个应用程序。很可能今天大多数在职程序员都在使用面向对象语言。他们不可能都错得那么离谱。

特别是，对于动态类型语言来说，对象是非常方便的。我们需要某种方式来定义复合数据类型，用来将一堆数据组合在一起。

如果我们也能把方法挂在这些对象上，那么我们就不需要把函数操作的数据类型的名字作为函数名称的前缀，以避免与不同类型的类似函数发生冲突。比如说，在 Racket 中，你最终不得不将你的函数命名为 hash-copy(复制一个哈希表) 和 vector-copy(复制一个向量)，这样它们就不会互相覆盖。方法的作用域是对象，所以这个问题就不存在了。

3.9.2 为什么 Lox 是面向对象的？

我可以声明对象是 groovy 的，但仍然超出了本书的范围。大多数编程语言的书籍，特别是那些试图实现一门完整语言的书籍，都忽略了对象。对我来说，这意味着这个主题没有被很好地覆盖。对于如此广泛使用的范式，这种遗漏让我感到悲伤。

鉴于我们很多人整天都在使用 OOP 语言，似乎这个世界应该有一些关于如何制作 OOP 语言的文档。正如你将看到的那样，事实证明这很有趣。没有你担心的那么难，但也没有你想象的那么简单。

3.9.3 类还是原型？

当涉及对象时，实际上有两种方法，类和原型。类最先出现，由于 C++、Java、C# 和其它近似语言的出现，类更加普遍。直到 JavaScript 意外地占领了世界之前，原型几乎

是一个被遗忘的分支。

在基于类的语言中，有两个核心概念：实例和类。实例存储每个对象的状态，并有一个对实例的类的引用。类包含方法和继承链。要在实例上调用方法，总是存在一个中间层。您要先查找实例的类，然后在其中找到方法：

基于原型的语言融合了这两个概念。这里只有对象——没有类，而且每个对象都可以包含状态和方法。对象之间可以直接继承（或者用原型语言的术语说是“委托”）：

这意味着原型语言在某些方面比类更基础。它们实现起来真的很整洁，因为它们很简单。另外，它们还可以表达很多不寻常的模式，而这些模式是类所不具备的。

但是我看过很多用原型语言写的代码——包括我自己设计的一些代码。你知道人们一般会怎么使用原型的强大功能和灵活性吗？... 他们用它来重新发明类。

我不知道这是为什么，但人们自然而然地似乎更喜欢基于类的（经典？优雅？）风格。原型在语言中更简单，但它们似乎只是通过将复杂性推给用户来实现的。所以，对于 Lox 来说，我们将省去用户的麻烦，直接把类包含进去。

3.9.4 Lox 中的类

理由已经说够了，来看看我们实际上拥有什么。在大多数语言中，类包含了一系列的特性。对于 Lox，我选择了我认为最闪亮的一点。您可以像这样声明一个类及其方法：

```
class Breakfast {
  cook() {
    print "Eggs a-fryin'!";
  }

  serve(who) {
    print "Enjoy your breakfast, " + who + ".";
  }
}
```

类的主体包含其方法。它们看起来像函数声明，但没有 fun 关键字。当类声明生效时，Lox 将创建一个类对象，并将其存储在以该类命名的变量中。就像函数一样，类在 Lox 中也是一等公民：

```
// Store it in variables.
var someVariable = Breakfast;

// Pass it to functions.
someFunction(Breakfast);
```

接下来，我们需要一种创建实例的方法。我们可以添加某种 new 关键字，但为了简单起见，在 Lox 中，类本身是实例的工厂函数。像调用函数一样调用一个类，它会生成

一个自己的新实例：

```
var breakfast = Breakfast();
print breakfast; // "Breakfast instance".
```

3.9.5 实例化和初始化

只有行为的类不是非常有用。面向对象编程背后的思想是将行为和状态封装在一起。为此，您需要有字段。Lox 和其他动态类型语言一样，允许您自由地向对象添加属性：

```
breakfast.meat = "sausage";
breakfast.bread = "sourdough";
```

如果一个字段不存在，那么对它进行赋值时就会先创建。

如果您想从方法内部访问当前对象上的字段或方法，可以使用 this：

```
class Breakfast {
  serve(who) {
    print "Enjoy your " + this.meat + " and " +
        this.bread + ", " + who + ".";
  }

  // ...
}
```

在对象中封装数据的目的之一是确保对象在创建时处于有效状态。为此，你可以定义一个初始化器。如果您的类中包含一个名为 init() 的方法，则在构造对象时会自动调用该方法。传递给类的任何参数都会转发给它的初始化器：

```
class Breakfast {
  init(meat, bread) {
    this.meat = meat;
    this.bread = bread;
  }

  // ...
}

var baconAndToast = Breakfast("bacon", "toast");
baconAndToast.serve("Dear Reader");
// "Enjoy your bacon and toast, Dear Reader."
```

3.9.6 继承

在每一种面向对象的语言中，你不仅可以定义方法，而且可以在多个类或对象中重用它们。为此，Lox 支持单继承。当你声明一个类时，你可以使用小于 (<) 操作符指定它继承的类：

```
class Brunch < Breakfast {
  drink() {
    print "How about a Bloody Mary?";
  }
}
```

这里，Brunch 是**派生类**或**子类**，而 Breakfast 是**基类**或**超类**。父类中定义的每个方法对其子类也可用：

```
var benedict = Brunch("ham", "English muffin");
benedict.serve("Noble Reader");
```

即使是 init() 方法也会被继承。在实践中，子类通常也想定义自己的 init() 方法。但还需要调用原始的初始化方法，以便超类能够维护其状态。我们需要某种方式能够调用自己实例上的方法，而无需触发实例自身的方法。

与 Java 中一样，您可以使用 super：

```
class Brunch < Breakfast {
  init(meat, bread, drink) {
    super.init(meat, bread);
    this.drink = drink;
  }
}
```

这就是面向对象的内容。我尽量将功能设置保持在最低限度。本书的结构确实迫使我做了一个妥协。Lox 不是一种纯粹的面向对象的语言。在真正的 OOP 语言中，每个对象都是一个类的实例，即使是像数字和布尔值这样的基本类型。

因为我们开始使用内置类型很久之后才会实现类，所以这一点很难实现。因此，从类实例的意义上说，基本类型的值并不是真正的对象。它们没有方法或属性。如果以后我想让 Lox 成为真正的用户使用的语言，我会解决这个问题。

3.10 标准库

我们快结束了，这就是整个语言，所剩下的就是“核心”或“标准”库——这是一组直接在解释器中实现的功能集，所有用户定义的行为都是建立在此之上。

这是 Lox 中最可悲的部分。它的标准库已经超过了极简主义，解决彻底的虚无主义。对于本书中的示例代码，我们只需要证明代码在运行，并且在做它应该做的事。为此，我

们已经有了内置的 print 语句。

稍后，当我们开始优化时，我们将编写一些基准测试，看看执行代码需要多长时间。这意味着我们需要跟踪时间，因此我们将定义一个内置函数 clock()，该函数会返回程序启动后的秒数。

嗯... 就是这样。我知道，有点尴尬，对吧？

如果您想将 Lox 变成一门实际可用的语言，那么您应该做的第一件事就是对其充实。字符串操作、三角函数、文件 I/O、网络、扩展，甚至读取用户的输入都将有所帮助。但对于本书来说，我们不需要这些，而且加入这些也不会教给你任何有趣的东西，所以我把它省略了。

别担心，这门语言本身就有很多精彩的内容让我们忙个不停。

3.11 习题

1. 编写一些示例 Lox 程序并运行它们 (您可以使用我的 Lox 实现)。试着想出我在这里没有详细说明的边界情况。它是否按照期望运行？为什么？
2. 这种非正式的介绍留下了很多未说明的东西。列出几个关于语言语法和语义的开放问题。你认为答案应该是什么？
3. Lox 是一种很小的语言。您认为缺少哪些功能会使其不适用于实际程序？(当然，除了标准库。)

3.12 设计笔记：表达式和语句

Lox 既有表达式也有语句。有些语言省略了后者。相对地，它们将声明和控制流结构也视为表达式。这类“一切都是表达式”的语言往往具有函数式的血统，包括大多数 Lisp、SML、Haskell、Ruby 和 CoffeeScript。

要做到这一点，对于语言中的每一个“类似于语句”的构造，你需要决定它所计算的值是什么。其中有些很简单：

- if 表达式的计算结果是所选分支的结果。同样，switch 或其他多路分支的计算结果取决于所选择的情况。
- 变量声明的计算结果是变量的值。
- 块的计算结果是序列中最后一个表达式的结果。

有一些是比较复杂的。循环应该计算什么值？在 CoffeeScript 中，一个 while 循环计算结果为一个数组，其中包含了循环体中计算到的每个元素。这可能很方便，但如果你不需要这个数组，就会浪费内存。

您还必须决定这些类似语句的表达式如何与其他表达式组合，必须将它们放入语法的优先表中。例如，Ruby 允许下面这种写法：

```
puts 1 + if true then 2 else 3 end + 4
```

这是你所期望的吗？这是你的用户所期望的吗？这对你如何设计“语句”的语法有什么影响？请注意，Ruby 有一个显式的 end 关键字来表明 if 表达式结束。如果没有它，+4 很可能会被解析为 else 子句的一部分。

把每个语句都转换成表达式会迫使你回答一些类似这样的复杂问题。作为回报，您消除了一些冗余。C 语言中既有用于排序语句的块，以及用于排序表达式的逗号操作符。它既有 if 语句，也有?: 条件操作符。如果在 C 语言中所有东西都是表达式，你就可以把它们统一起来。

取消了语句的语言通常还具有**隐式返回**的特点——函数自动返回其函数主体所计算得到的任何值，而不需要显式的 return 语法。对于小型函数和方法来说，这真的很方便。事实上，许多有语句的语言都添加了类似于 => 的语法，以便能够定义函数体是计算单一表达式结果的函数。

但是让所有的函数以这种方式工作可能有点奇怪。即使你只是想让函数产生副作用，如果不小心，函数也可能会泄露返回值。但实际上，这些语言的用户并不觉得这是一个问题。

对于 Lox，我在其中添加语句是出于朴素的原因。为了熟悉起见，我选择了一种类似于 C 的语法，而试图把现有的 C 语句语法像表达式一样解释，会变得非常快。

第二部分

树遍历解释器

第四章 扫描

> 大干特工。每件值得做的事都值得反复去做。
>
> ——罗伯特

任何编译器或解释器的第一步都是扫描。扫描器以一系列字符的形式接收原始源代码，并将其分组成一系列的块，我们称之为**标识**（词法单元）。这些是有意义的“单词”和“标点”，它们构成了语言的语法。

对于我们来说，扫描也是一个很好的起点，因为代码不是很难——相当于有很多分支的 switch 语句。这可以帮助我们在学习更后面有趣的部分之前进行热身。在本章结束时，我们将拥有一个功能齐全、速度快的扫描器，它可以接收任何一串 Lox 源代码，并产生标记，我们将在下一章把这些标记输入到解析器中。

4.1 解释器框架

由于这是我们的第一个真正的章节，在我们开始实际扫描代码之前，我们需要先勾勒出我们的解释器 jlox 的基本形态。在 Java 中，一切都是从一个类开始的。

lox/Lox.java，创建新文件

```
package com.craftinginterpreters.lox;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.nio.charset.Charset;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.List;

public class Lox {
  public static void main(String[] args) throws IOException {
    if (args.length > 1) {
      System.out.println("Usage: jlox [script]");
      System.exit(64);
    } else if (args.length == 1) {
      runFile(args[0]);
    } else {
      runPrompt();
    }
  }
}
```

把它贴在一个文本文件里，然后去把你的 IDE 或者 Makefile 或者其他工具设置好。我就在这里等你准备好。好了吗？好的！

Lox 是一种脚本语言，这意味着它直接从源代码执行。我们的解释器支持两种运行代码的方式。如果从命令行启动 jlox 并为其提供文件路径，它将读取该文件并执行。

lox/Lox.java，添加到 main() 方法之后

```
private static void runFile(String path) throws IOException {
  byte[] bytes = Files.readAllBytes(Paths.get(path));
  run(new String(bytes, Charset.defaultCharset()));
}
```

lox/Lox.java，添加到 runFile() 方法之后

```
private static void runPrompt() throws IOException {
  InputStreamReader input = new InputStreamReader(System.in);
  BufferedReader reader = new BufferedReader(input);

  for (;;) {
    System.out.print("> ");
    String line = reader.readLine();
    if (line == null) break;
    run(line);
  }
}
```

readLine() 函数，顾名思义，读取用户在命令行上的一行输入，并返回结果。要终止交互式命令行应用程序，通常需要输入 Control-D。这样做会向程序发出“文件结束”的信号。当这种情况发生时，readLine() 就会返回 null，所以我们检查一下是否存在 null 以退出循环。

交互式提示符和文件运行工具都是对这个核心函数的简单包装：

lox/Lox.java，添加到 runPrompt() 之后

```
  private static void run(String source) {
    Scanner scanner = new Scanner(source);
    List<Token> tokens = scanner.scanTokens();

    // For now, just print the tokens.
    for (Token token : tokens) {
      System.out.println(token);
    }
  }
```

因为我们还没有写出解释器，所以这些代码还不是很有用，但这只是小步骤，你要明白？现在，它可以打印出我们即将完成的扫描器所返回的标记，这样我们就可以看到我们的解析是否生效。

4.1.1 错误处理

当我们设置东西的时候，另一个关键的基础设施是错误处理。教科书有时会掩盖这一点，因为这更多的是一个实际问题，而不是一个正式的计算机科学问题。但是，如果你关心的是如何制作一个真正可用的语言，那么优雅地处理错误是至关重要的。

我们的语言提供的处理错误的工具构成了其用户界面的很大一部分。当用户的代码

在工作时，他们根本不会考虑我们的语言——他们的脑子里都是他们的程序。通常只有当程序出现问题时，他们才会注意到我们的实现。

当这种情况发生时，我们就需要向用户提供他们所需要的所有信息，让他们了解哪里出了问题，并引导他们慢慢达到他们想要去的地方。要做好这一点，意味着从现在开始，在解释器的整个实现过程中都要考虑错误处理。

lox/Lox.java，添加到 run() 方法之后

```
static void error(int line, String message) {
  report(line, "", message);
 }

 private static void report(int line, String where,
                            String message) {
  System.err.println(
    "[line " + line + "] Error" + where + ": " + message);
  hadError = true;
 }
```

这个 error() 函数和其工具方法 report() 会告诉用户在某一行上发生了一些语法错误。这其实是最起码的，可以说你有错误报告功能。想象一下，如果你在某个函数调用中不小心留下了一个悬空的逗号，解释器就会打印出来：

```
Error: Unexpected "," somewhere in your code. Good luck finding it!
```

这种信息没有多大帮助。我们至少要给他们指出正确的方向。好一些的做法是指出开头和结尾一栏，这样他们就知道这一行的位置了。更好的做法是向用户显示违规的行，比如：

```
Error: Unexpected "," in argument list.

    15 | function(first, second,);
                               ^-- Here.
```

我很想在这本书里实现这样的东西，但老实说，这会引入很多繁琐的字符串操作代码。这些代码对用户来说非常有用，但在书中读起来并不友好，而且技术上也不是很有趣。所以我们还是只用一个行号。在你们自己的解释器中，请按我说的做，而不是按我做的做。

我们在 Lox 主类中坚持使用这个错误报告功能的主要原因就是因为那个 hadError 字段。它的定义在这里：

lox/Lox.java 在 Lox 类中添加：

```
public class Lox {
  static boolean hadError = false;
```

我们将以此来确保我们不会尝试执行有已知错误的代码。此外，它还能让我们像一个好的命令行工具那样，用一个非零的结束代码退出。

lox/Lox.java，在 runFile() 中添加：

```
    run(new String(bytes, Charset.defaultCharset()));

    // Indicate an error in the exit code.
    if (hadError) System.exit(65);
  }
```

我们需要在交互式循环中重置此标志。如果用户输入有误，也不应终止整个会话。

lox/Lox.java，在 runPrompt() 中添加：

```
      run(line);
      hadError = false;
    }
```

我把错误报告拉出来，而不是把它塞进扫描器和其他可能发生错误的阶段，还有另一个原因，是为了提醒您，把产生错误的代码和报告错误的代码分开是一个很好的工程实践。

前端的各个阶段都会检测到错误，但是它们不需要知道如何向用户展示错误。再一个功能齐全的语言实现中，可能有多种方式展示错误信息：在 stderr，在 IDE 的错误窗口中，记录到文件，等等。您肯定不希望扫描器和解释器中到处充斥着这类代码。

理想情况下，我们应该有一个实际的抽象，即传递给扫描程序和解析器的某种 ErrorReporter 接口，这样我们就可以交换不同的报告策略。对于我们这里的简单解释器，我没有那样做，但我至少将错误报告代码移到了一个不同的类中。

有了一些基本的错误处理，我们的应用程序外壳已经准备好了。一旦我们有了一个带有 scanTokens() 方法的 Scanner 类，我们就可以开始运行它了。在我们开始之前，让我们更精确地了解什么是标记（tokens）。

4.2 词素和标记（词法单元）

下面是一行 lox 代码：

```
var language = "lox";
```

在这里，var 是声明变量的关键字。“v-a-r” 这三个字符的序列是有意义的。但如果我们从 language 中间抽出三个字母，比如 “g-u-a”，它们本身并没有任何意义。

这就是词法分析的意义所在。我们的工作是扫描字符列表，并将它们归纳为具有某些含义的最小序列。每一组字符都被称为词素。在示例代码行中，词素是：

词素只是源代码的原始子字符串。但是，在将字符序列分组为词素的过程中，我们也会发现了一些其他有用的信息。当我们获取词素并将其与其他数据捆绑在一起时，结果是一个标记（token，词法单元）。它包含一些有用的内容，比如：

4.2.1 标记类型

关键词是语言语法的一部分，所以解析器经常会有这样的代码："如果下一个标记是while，那么就....."。这意味着解析器想知道的不仅仅是它有某个标识符的词素，而是它得到一个保留词，以及它是哪个关键词。

解析器可以通过比较字符串对原始词素中的标记进行分类，但这样做很慢，而且有点难看。相反，在我们识别一个词素的时候，我们还要记住它代表的是哪种词素。我们为每个关键字、操作符、标点位和字面量都有不同的类型。

lox/TokenType.java，创建新文件

```
package com.craftinginterpreters.lox;

enum TokenType {
  // Single-character tokens.
  LEFT_PAREN, RIGHT_PAREN, LEFT_BRACE, RIGHT_BRACE,
  COMMA, DOT, MINUS, PLUS, SEMICOLON, SLASH, STAR,

  // One or two character tokens.
  BANG, BANG_EQUAL,
  EQUAL, EQUAL_EQUAL,
  GREATER, GREATER_EQUAL,
  LESS, LESS_EQUAL,

  // Literals.
  IDENTIFIER, STRING, NUMBER,

  // Keywords.
  AND, CLASS, ELSE, FALSE, FUN, FOR, IF, NIL, OR,
  PRINT, RETURN, SUPER, THIS, TRUE, VAR, WHILE,

  EOF
}
```

4.2.2 字面量

字面量有对应词素——数字和字符串等。由于扫描器必须遍历文字中的每个字符才能正确识别，所以它还可以将值的文本表示转换为运行时对象，解释器后续将使用该对象。

4.2.3 位置信息

早在我宣讲错误处理的福音时，我们就看到，我们需要告诉用户错误发生在哪里。（用户）从这里开始定位问题。在我们的简易解释器中，我们只说明了标记出现在哪一行上，但更复杂的实现中还应该包括列位置和长度。

我们将所有这些数据打包到一个类中。

lox/Token.java，创建新文件

```
package com.craftinginterpreters.lox;

class Token {
  final TokenType type;
  final String lexeme;
  final Object literal;
  final int line;

  Token(TokenType type, String lexeme, Object literal, int line) {
    this.type = type;
    this.lexeme = lexeme;
    this.literal = literal;
    this.line = line;
  }

  public String toString() {
    return type + " " + lexeme + " " + literal;
  }
}
```

现在我们有了一个信息充分的对象，足以支撑解释器的所有后期阶段。

4.3 正则语言和表达式

既然我们已知道我们要输出的什么，那么，我们就开始吧。扫描器的核心是一个循环。从源码的第一个字符开始，扫描器计算出该字符属于哪个词素，并消费它和属于该词素的任何后续字符。当到达该词素的末尾时，扫描器会输出一个令牌。

然后再循环一次，它又循环回来，从源代码中的下一个字符开始再做一次。它一直这样做，吃掉字符，偶尔，呃，排出令牌，直到它到达输入的终点。

在循环中，我们会查看一些字符，以确定它“匹配”的是哪种词素，这部分内容可能听起来很熟悉，但如果你知道正则表达式，你可以考虑为每一种词素定义一个 regex，并使用这些 regex 来匹配字符。如果你了解正则表达式，你可以考虑为每一种词素定义一个 regex，然后用来匹配字符。例如，Lox 对标识符（变量名等）的规则与 C 语言相同。下面的 regex 可以匹配一个标识符：

```
[a-zA-Z_][a-zA-Z_0-9]*
```

如果你确实想到了正则表达式，那么你的直觉还是很深刻的。决定一门语言如何将字符分组为词素的规则被称为它的**词法语法**。在 Lox 中，和大多数编程语言一样，该语法的规则非常简单，可以将其归为**正则语言**。这里的正则和正则表达式中的“正则”是一样的含义。

如果你愿意，你可以非常精确地使用正则表达式来识别 Lox 的所有不同词组，而且还有一堆有趣的理论来支撑着为什么会这样以及它的意义。像 Lex 或 Flex 这样的工具就是专门为实现这一功能而设计的——向其中传入一些正则表达式，它可以为您提供完整的扫描器。

由于我们的目标是了解扫描器是如何工作的，所以我们不会把这个任务交给正则表达式。我们要亲自动手实现。

4.4 Scanner 类

事不宜迟，我们先来建一个扫描器吧。

lox/Scanner.java，创建新文件

```
package com.craftinginterpreters.lox;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import static com.craftinginterpreters.lox.TokenType.*;

class Scanner {
 private final String source;
 private final List<Token> tokens = new ArrayList<>();

 Scanner(String source) {
  this.source = source;
 }
}
```

我们将原始的源代码存储为一个简单的字符串，并且我们已经准备了一个列表来保存扫描时产生的标记。前面提到的循环看起来类似于：

lox/Scanner.java，方法 Scanner() 后添加

```
 List<Token> scanTokens() {
  while (!isAtEnd()) {
   // We are at the beginning of the next lexeme.
   start = current;
   scanToken();
  }

  tokens.add(new Token(EOF, "", null, line));
  return tokens;
 }
```

扫描器通过自己的方式遍历源代码，添加标记，直到遍历完所有字符。然后，它在最后附加一个的“end of file”标记。严格意义上来说，这并不是必须的，但它可以使我们的解析器更加干净。

这个循环依赖于几个字段来跟踪扫描器在源代码中的位置。

<u></u>

lox/Scanner.java，在 Scanner 类中添加：

```
private final List<Token> tokens = new ArrayList<>();
private int start = 0;
private int current = 0;
private int line = 1;

Scanner(String source) {
```

start 和 current 字段是指向字符串的偏移量。start 字段指向被扫描的词素中的第一个字符，current 字段指向当前正在处理的字符。line 字段跟踪的是 current 所在的源文件行数，这样我们产生的标记就可以知道其位置。

然后，我们还有一个辅助函数，用来告诉我们是否已消费完所有字符。

lox/Scanner.java 在 scanTokens() 方法之后添加：

```
private boolean isAtEnd() {
 return current >= source.length();
}
```

4.5 识别词素

在每一次循环中，我们会扫描一个标记。这是扫描器真正的核心。让我们先从简单情况开始。想象一下，如果每个词素只有一个字符长。您所需要做的就是消费下一个字符并为其选择一个标记类型。在 Lox 中有一些词素只包含一个字符，所以我们从这些词素开始。

lox/Scanner.java 添加到 scanTokens() 方法之后

```
private void scanToken() {
  char c = advance();
  switch (c) {
    case '(': addToken(LEFT_PAREN); break;
    case ')': addToken(RIGHT_PAREN); break;
    case '{': addToken(LEFT_BRACE); break;
    case '}': addToken(RIGHT_BRACE); break;
    case ',': addToken(COMMA); break;
    case '.': addToken(DOT); break;
    case '-': addToken(MINUS); break;
    case '+': addToken(PLUS); break;
    case ';': addToken(SEMICOLON); break;
    case '*': addToken(STAR); break;
  }
}
```

同样，我们也需要一些辅助方法。

lox/Scanner.java，添加到 isAtEnd() 方法后

```
private char advance() {
  current++;
  return source.charAt(current - 1);
}

private void addToken(TokenType type) {
  addToken(type, null);
}

private void addToken(TokenType type, Object literal) {
  String text = source.substring(start, current);
  tokens.add(new Token(type, text, literal, line));
}
```

advance() 方法获取源文件中的下一个字符并返回它。advance() 用于处理输入，addToken() 则用于输出。该方法获取当前词素的文本并为其创建一个新标记。我们马上会使用另一个重载方法来处理带有字面值的标记。

4.5.1 词法错误

在我们深入探讨之前，我们先花一点时间考虑一下词法层面的错误。如果用户抛入解释器的源文件中包含一些 Lox 中不使用的字符——如 @#^会发生什么？现在，这些字符被默默抛弃了。它们没有被 Lox 语言使用，但是不意味着解释器可以假装它们不存在。相反，我们应该报告一个错误：

lox/Scanner.java 在 scanToken() 方法中添加：

```
      case '*': addToken(STAR); break;

      default:
        Lox.error(line, "Unexpected character.");
        break;
    }
```

注意，错误的字符仍然会被前面调用的 advance() 方法消费。这一点很重要，这样我们就不会陷入无限循环了。

另请注意，我们一直在扫描。程序稍后可能还会出现其他错误。如果我们能够一次检测出尽可能多的错误，将为我们的用户带来更好的体验。否则，他们会看到一个小错误并修复它，但是却出现下一个错误，不断重复这个过程。语法错误“打地鼠”一点也不好玩。

(别担心。因为 hadError 进行了赋值，我们永远不会尝试执行任何代码，即使程序在继续运行并扫描代码文件的其余部分。)

4.5.2 运算符

我们的单字符词素已经生效了，但是这不能涵盖 Lox 中的所有操作符。比如!? 这是单字符，对吧？有时候是的，但是如果下一个字符是等号，那么我们应该改用!= 词素。注意，这里的! 和 = 不是两个独立的运算符。在 Lox 中，你不能写! = 来表示不等操作符。这就是为什么我们需要将!= 作为单个词素进行扫描。同样地，<、> 和 = 都可以与后面跟随的 = 来组合成其他相等和比较操作符。

对于所有这些情况，我们都需要查看第二个字符。

lox/Scanner.java，在 scanToken() 方法中添加

```
      case '*': addToken(STAR); break;
      case '!':
        addToken(match('=') ? BANG_EQUAL : BANG);
        break;
      case '=':
        addToken(match('=') ? EQUAL_EQUAL : EQUAL);
        break;
      case '<':
        addToken(match('=') ? LESS_EQUAL : LESS);
        break;
      case '>':
        addToken(match('=') ? GREATER_EQUAL : GREATER);
        break;
      default:
```

这些分支中使用了下面的新方法：

lox/Scanner.java 添加到 scanToken() 方法后

```
  private boolean match(char expected) {
    if (isAtEnd()) return false;
    if (source.charAt(current) != expected) return false;

    current++;
    return true;
  }
```

这就像一个有条件的 advance()。只有当前字符是我们正在寻找的字符时，我们才会消费。

使用 match()，我们分两个阶段识别这些词素。例如，当我们得到! 时，我们会跳转到它的 case 分支。这意味着我们知道这个词素是以! 开始的。然后，我们查看下一个字符，以确认词素是一个!= 还是仅仅是一个!。

4.6 更长的词素

我们还缺少一个操作符：表示除法的/。这个字符需要一些特殊处理，因为注释也是以斜线开头的。

lox/Scanner.java，在 scanToken() 方法中添加：

```
      break;
      case '/':
        if (match('/')) {
          // A comment goes until the end of the line.
          while (peek() != '\n' && !isAtEnd()) advance();
        } else {
          addToken(SLASH);
        }
        break;
      default:
```

这与其它的双字符运算符是类似的，区别在于我们找到第二个/时，还没有结束本次标记。相反，我们会继续消费字符直至行尾。

这是我们处理较长词素的一般策略。当我们检测到一个词素的开头后，我们会分流到一些特定于该词素的代码，这些代码会不断地消费字符，直到结尾。

我们又有了一个辅助函数：

lox/Scanner.java，在 match() 方法后添加：

```
  private char peek() {
    if (isAtEnd()) return '\0';
    return source.charAt(current);
  }
```

这有点像 advance() 方法，只是不会消费字符。这就是所谓的 **lookahead(前瞻)**。因为它只关注当前未消费的字符，所以我们有一个前瞻字符。一般来说，数字越小，扫描器运行速度就越快。词法语法的规则决定了我们需要前瞻多少字符。幸运的是，大多数广泛使用的语言只需要提前一到两个字符。

注释是词素，但是它们没有含义，而且解析器也不想要处理它们。所以，我们达到注释末尾后，不会调用 addToken() 方法。当我们循环处理下一个词素时，start 已经被重置了，注释的词素就消失在一阵烟雾中了。

既然如此，现在正好可以跳过其它那些无意义的字符了：换行和空格。

lox/Scanner.java，在 scanToken() 方法中添加：

```
      break;
    case ' ':
    case '\r':
    case '\t':
      // Ignore whitespace.
      break;

    case '\n':
      line++;
      break;
    default:
      Lox.error(line, "Unexpected character.");
```

当遇到空白字符时，我们只需回到扫描循环的开头。这样就会在空白字符之后开始一个新的词素。对于换行符，我们做同样的事情，但我们也会递增行计数器。（这就是为什么我们使用 peek() 而不是 match() 来查找注释结尾的换行符。我们到这里希望能读取到换行符，这样我们就可以更新行数了）

我们的扫描器越来越聪明了。它可以处理相当自由形式的代码，如：

```
// this is a comment
(( )){} // grouping stuff
!*+-/=<> <= == // operators
```

4.6.1 字符串字面量

现在我们对长词素已经很熟悉了，我们可以开始处理字面量了。我们先处理字符串，因为字符串总是以一个特定的字符" 开头。

lox/Scanner.java，在 scanToken() 方法中添加：

```
      break;
    case '"': string(); break;
    default:
```

这里会调用：

lox/Scanner.java，在 scanToken() 方法之后添加：

```
private void string() {
  while (peek() != '"' && !isAtEnd()) {
    if (peek() == '\n') line++;
    advance();
  }

  if (isAtEnd()) {
    Lox.error(line, "Unterminated string.");
    return;
  }

  // The closing ".
  advance();

  // Trim the surrounding quotes.
  String value = source.substring(start + 1, current - 1);
  addToken(STRING, value);
}
```

与注释类似，我们会一直消费字符，直到" 结束该字符串。如果输入内容耗尽，我们也会进行优雅的处理，并报告一个对应的错误。

没有特别的原因，Lox 支持多行字符串。这有利有弊，但禁止换行比允许换行更复杂一些，所以我把它们保留了下来。这意味着当我们在字符串内遇到新行时，我们也需要更新 line 值。

最后，还有一个有趣的地方就是当我们创建标记时，我们也会产生实际的字符串值，该值稍后将被解释器使用。这里，值的转换只需要调用 substring() 剥离前后的引号。如果 Lox 支持转义序列，比如
n，我们会在这里取消转义。

4.6.2 数字字面量

在 Lox 中，所有的数字在运行时都是浮点数，但是同时支持整数和小数字面量。一个数字字面量就是一系列数位，后面可以跟一个. 和一或多个尾数。

```
1234
12.34
```

我们不允许小数点处于最开始或最末尾，所以下面的格式是不正确的：

```
.1234
1234.
```

我们可以很容易地支持前者，但为了保持简单，我把它删掉了。如果我们要允许对数字进行方法调用，比如 123.sqrt()，后者会变得很奇怪。

为了识别数字词素的开头，我们会寻找任何一位数字。为每个十进制数字添加 case 分支有点乏味，所以我们直接在默认分支中进行处理。

lox/Scanner.java，在 scanToken() 方法中替换 1 行：

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

这里依赖下面的小工具函数：

lox/Scanner.java，在 peek() 方法之后添加：

```
  private boolean isDigit(char c) {
    return c >= '0' && c <= '9';
  }
```

一旦我们知道当前在处理数字，我们就分支进入一个单独的方法消费剩余的字面量，跟字符串的处理类似。

lox/Scanner.java，在 scanToken() 方法后添加：

```
private void number() {
  while (isDigit(peek())) advance();

  // Look for a fractional part.
  if (peek() == '.' && isDigit(peekNext())) {
    // Consume the "."
    advance();

    while (isDigit(peek())) advance();
  }

  addToken(NUMBER,
      Double.parseDouble(source.substring(start, current)));
}
```

我们在字面量的整数部分中尽可能多地获取数字。然后我们寻找小数部分，也就是一个小数点 (.) 后面至少跟一个数字。如果确实有小数部分，同样地，我们也尽可能多地获取数字。

在定位到小数点之后需要前瞻两个字符，因为我们只有确认其后有数字才会消费.。所以我们添加了：

lox/Scanner.java，在 peek() 方法后添加

```
private char peekNext() {
  if (current + 1 >= source.length()) return '\0';
  return source.charAt(current + 1);
}
```

最后，我们将词素转换为其对应的数值。我们的解释器使用 Java 的 Double 类型来表示数字，所以我们创建一个该类型的值。我们使用 Java 自带的解析方法将词素转换为真正的 Java double。我们可以自己实现，但是，说实话，除非你想为即将到来的编程面试做准备，否则不值得你花时间。

剩下的词素是 Boolean 和 nil，但我们把它们作为关键字来处理，这样我们就来到了.....

4.7 保留字和标识符

我们的扫描器基本完成了，词法语法中还需要实现的部分仅剩标识符及其近亲——保留字。你也许会想，我们可以采用与处理 <= 等多字符操作符时相同的方法来匹配关键字，如 or。

```
case 'o':
 if (peek() == 'r') {
  addToken(OR);
 }
 break;
```

考虑一下，如果用户将变量命名为 orchid 会发生什么？扫描器会先看到前面的两个字符，然后立刻生成一个 or 标记。这就涉及到了一个重要原则，叫作 **maximal munch**(最长匹配)。当两个语法规则都能匹配扫描器正在处理的一大块代码时，哪个规则相匹配的字符最多，就使用哪个规则。

该规则规定，如果我们可以将 orchid 匹配为一个标识符，也可以将 or 匹配为一个关键字，那就采用第一种结果。这也就是为什么我们在前面会默认为，<= 应该识别为单一的 <= 标记，而不是 < 后面跟了一个 =。

最大匹配原则意味着，我们只有扫描完一个可能是标识符的片段，才能确认是否一个保留字。毕竟，保留字也是一个标识符，只是一个已经被语言要求为自己所用的标识符。这也是**保留字**一词的由来。

所以我们首先假设任何以字母或下划线开头的词素都是一个标识符。

lox/Scanner.java，在 scanToken() 中添加代码

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

其它代码如下：

lox/Scanner.java，在 scanToken() 方法之后添加：

```
 private void identifier() {
  while (isAlphaNumeric(peek())) advance();

  addToken(IDENTIFIER);
 }
```

通过以下辅助函数来定义：

lox/Scanner.java，在 peekNext() 方法之后添加：

```
private boolean isAlpha(char c) {
 return (c >= 'a' && c <= 'z')
      (c >= 'A' && c <= 'Z')
       c == '_';
}

private boolean isAlphaNumeric(char c) {
 return isAlpha(c)  isDigit(c);
}
```

这样标识符就开始工作了。为了处理关键字，我们要查看标识符的词素是否是保留字之一。如果是，我们就使用该关键字特有的标记类型。我们在 map 中定义保留字的集合。

lox/Scanner.java，在 Scanner 类中添加：

```
private static final Map<String, TokenType> keywords;

static {
 keywords = new HashMap<>();
 keywords.put("and",    AND);
 keywords.put("class",  CLASS);
 keywords.put("else",   ELSE);
 keywords.put("false",  FALSE);
 keywords.put("for",    FOR);
 keywords.put("fun",    FUN);
 keywords.put("if",     IF);
 keywords.put("nil",    NIL);
 keywords.put("or",     OR);
 keywords.put("print",  PRINT);
 keywords.put("return", RETURN);
 keywords.put("super",  SUPER);
 keywords.put("this",   THIS);
 keywords.put("true",   TRUE);
 keywords.put("var",    VAR);
 keywords.put("while",  WHILE);
}
```

接下来，在我们扫描到标识符之后，要检查是否与 map 中的某些项匹配。

lox/Scanner.java，在 identifier() 方法中替换一行：

```
    while (isAlphaNumeric(peek())) advance();

    String text = source.substring(start, current);
    TokenType type = keywords.get(text);
    if (type == null) type = IDENTIFIER;
    addToken(type);
  }
```

如果匹配的话，就使用关键字的标记类型。否则，就是一个普通的用户定义的标识符。

至此，我们就有了一个完整的扫描器，可以扫描整个 Lox 词法语法。启动 REPL，输入一些有效和无效的代码。它是否产生了你所期望的词法单元？试着想出一些有趣的边界情况，看看它是否能正确地处理它们。

4.8 挑战

1. Python 和 Haskell 的语法不是正则的。这是什么意思，为什么不是呢？
   - Python 和 Haskell 都采用了对缩进敏感的语法，所以它们必须将缩进级别的变动识别为词法标记。这样做需要比较连续行的开头空格数量，这是使用常规语法无法做到的。
2. 除了分隔标记——区分 print foo 和 printfoo——空格在大多数语言中并没有什么用处。在 CoffeeScript、Ruby 和 C 预处理器中的一些隐秘的地方，空格确实会影响代码解析方式。在这些语言中，空格在什么地方，会有什么影响？
3. 我们这里的扫描器和大多数扫描器一样，会丢弃注释和空格，因为解析器不需要这些。什么情况下你会写一个不丢弃这些的扫描器？它有什么用呢？
4. 为 Lox 扫描器增加对 C 样式/ * ... * /屏蔽注释的支持。确保要处理其中的换行符。考虑允许它们嵌套，增加对嵌套的支持是否比你预期的工作更多？为什么？

4.9 设计笔记：隐藏的分号

现在的程序员已经被越来越多的语言选择宠坏了，对语法也越来越挑剔。他们希望自己的代码看起来干净、现代化。几乎每一种新语言都会放弃一个小的语法点（一些古老的语言，比如 BASIC 从来没有过），那就是将; 作为显式的语句结束符。

相对地，它们将“有意义的”换行符看作是语句结束符。这里所说的“有意义的”是有挑战性的部分。尽管大多数的语句都是在同一行，但有时你需要将一个语句扩展到多行。这些混杂的换行符不应该被视作结束符。

大多数明显的应该忽略换行的情况都很容易发现，但也有少数讨厌的情况：

- 返回值在下一行：

```
if (condition) return
"value"
```

“value”是要返回的值吗？还是说我们有一个空的 return 语句，后面跟着包含一个字符串字面量的表达式语句。

- 下一行中有带圆括号的表达式：

```
func
(parenthesized)
```

这是一个对 func(parenthesized) 的调用，还是两个表达式语句，一个用于 func，一个用于圆括号表达式？

- “-”号在下一行：

```
first
-second
```

这是一个中缀表达式——first - second，还是两个表达式语句，一个是 first，另一个是对 second 取负？

在所有这些情况下，无论是否将换行符作为分隔符，都会产生有效的代码，但可能不是用户想要的代码。在不同的语言中，有各种不同的规则来决定哪些换行符是分隔符。下面是几个例子：

- Lua 完全忽略了换行符，但是仔细地控制了它的语法，因此在大多数情况下，语句之间根本不需要分隔符。这段代码是完全合法的：

```
a = 1 b = 2
```

Lua 要求 return 语句是一个块中的最后一条语句，从而避免 return 问题。如果在关键字 end 之前、return 之后有一个值，这个值必须是用于 return。对于其他两种情况来说，Lua 允许显式的; 并且期望用户使用它。在实践中，这种情况基本不会发生，因为在小括号或一元否定表达式语句中没有任何意义。

- Go 会处理扫描器中的换行。如果在词法单元之后出现换行，并且该词法标记是已知可能结束语句的少数标记类型之一，则将换行视为分号，否则就忽略它。Go 团队提供了一个规范的代码格式化程序 gofmt，整个软件生态系统非常热衷于使用它，这确保了常用样式的代码能够很好地遵循这个简单的规则。
- Python 将所有换行符都视为有效，除非在行末使用明确的反斜杠将其延续到下一行。但是，括号（()、[] 或）内的任何换行都将被忽略。惯用的代码风格更倾向于后者。

这条规则对 Python 很有效，因为它是一种高度面向语句的语言。特别是，Python 的语法确保了语句永远不会出现在表达式内。C 语言也是如此，但许多其他有 lambda 或函数字面语法的语言则不然。

举一个 JavaScript 中的例子：

```
console.log(function() {
 statement();
});
```

这里，console.log() 表达式包含一个函数字面量，而这个函数字面量又包含 statement(); 语句。

如果要求进入一个嵌套在括号内的语句中，并且要求其中的换行是有意义的，那么 Python 将需要一套不同的隐式连接行的规则。

- JavaScript 的“自动分号插入”规则才是真正的奇葩。其他语言认为大多数换行符都是有意义的，只有少数换行符在多行语句中应该被忽略，而 JS 的假设恰恰相反。它将所有的换行符都视为无意义的空白，除非遇到解析错误。如果遇到了，它就会回过头来，尝试把之前的换行变成分号，以期得到正确的语法。

  如果我完全详细地介绍它是如何工作的，那么这个设计说明就会变成一篇设计檄文，更不用说 JavaScript 的“解决方案”从各种角度看都是个坏主意。真是一团糟。许多风格指南要求在每条语句后都显式地使用分号，尽管理论上该语言允许您省略它们，但 JavaScript 是我所知道的唯一一种（省略分号的）语言。

如果您要设计一种新的语言，则几乎可以肯定应该避免使用显式的语句终止符。程序员和其他人类一样是时尚的动物，分号和 ALL CAPS KEYWORDS(全大写关键字) 一样已经过时了。只是要确保您选择了一套适用于您语言的特定语法和习语的规则即可。不要重蹈 JavaScript 的覆辙。

第五章 表示代码

> 对于森林中的居民来说，几乎每一种树都有它的声音和特点。
>
> ——哈代

在上一章中，我们以字符串形式接收原始源代码，并将其转换为一个稍高级别的表示：一系列词法标记。我们在下一章中要编写的解析器，会将这些词法标记再次转换为更丰富、更复杂的表示形式。

在我们能够输出这种表示形式之前，我们需要先对其进行定义。这就是本章的主题。在这一过程中，我们将围绕形式化语法进行一些理论讲解，感受函数式编程和面向对象编程的区别，会介绍几种设计模式，并进行一些元编程。

在做这些事情之前，我们先关注一下主要目标——代码的表示形式。它应该易于解析器生成，也易于解释器使用。如果您还没有编写过解析器或解释器，那么这样的需求描述并不能很好地说明问题。也许你的直觉可以帮助你。当你扮演一个人类解释器的角色时，你的大脑在做什么？你如何在心里计算这样的算术表达式：

```
1 + 2 * 3 - 4
```

因为你已经理解了操作的顺序——以前的“Please Excuse My Dear Aunt Sally”之类，你知道乘法在加减操作之前执行。有一种方法可以将这种优先级进行可视化，那就是使用树。叶子节点是数字，内部节点是运算符，它们的每个操作数都对应一个分支。

要想计算一个算术节点，你需要知道它的子树的数值，所以你必须先计算子树的结果。这意味着要从叶节点一直计算到根节点——后序遍历：

- A. 从完整的树开始，先计算最下面的操作 2*3；
- B. 现在计算 +；
- C. 接下来，计算-；
- D. 最终得到答案。

如果我给你一个算术表达式，你可以很容易地画出这样的树；给你一棵树，你也可以毫不费力地进行计算。因此，从直观上看，我们的代码的一种可行的表示形式是一棵与语言的语法结构（运算符嵌套）相匹配的树。

那么我们需要更精确地了解这个语法是什么。就像上一章的词法词法一样，围绕句法语法也有一大堆理论。我们要比之前处理扫描时投入更多精力去研究这个理论，因为它在整个解释器的很多地方都是一个有用的工具。我们先从乔姆斯基谱系中往上升一级.....

5.1 上下文无关语法

在上一章中，我们用来定义词法语法（字符如何被分组为词法标记的规则）的形式体系，被称为*正则语言*。这对于我们的扫描器来说没什么问题，因为它输出的是一个扁

平的词法标记序列。但正则语言还不够强大，无法处理可以任意深度嵌套的表达式。

我们还需要一个更强大的工具，就是上下文无关语法 (**context-free grammar**，CFG)。它是形式化语法的工具箱中下一个最重的工具。一个形式化语法需要一组原子片段，它称之为“alphabet（字母表）”。然后它定义了一组（通常是无限的）“strings（字符串）”，这些字符串“包含”在语法中。每个字符串都是字母表中“letters（字符）”的序列。

我这里使用引号是因为当你从词法转到文法语法时，这些术语会让你有点困惑。在我们的扫描器词法中，alphabet（字母表）由单个字符组成，strings（字符串）是有效的词素（粗略的说，就是“单词”）。在现在讨论的文法语法中，我们处于一个不同的粒度水平。现在，字母表中的一个“letters（字符）”是一个完整的词法标记，而“strings（字符串）”是一个词法标记系列——一个完整的表达式。

嗯，使用表格可能更有助于理解：

术语		词法	语法
字母表	→	字符	词法标记
字符串	→	词素或词法标记	表达式
实现	→	扫描器	解析器

形式化语法的工作是指定哪些字符串有效，哪些无效。如果我们要为英语句子定义一个语法，“eggs are tasty for breakfast”会包含在语法中，但“tasty breakfast for are eggs”可能不会。

5.1.1 语法规则

我们如何写下一个包含无限多有效字符串的语法？我们显然无法一一列举出来。相反，我们创建了一组有限的规则。你可以把它们想象成一场你可以朝两个方向“玩”的游戏。

如果你从规则入手，你可以用它们生成语法中的字符串。以这种方式创建的字符串被称为**推导式**（派生式），因为每个字符串都是从语法规则中推导出来的。在游戏的每一步中，你都要选择一条规则，然后按照它告诉你的去做。围绕形式化语法的大部分语言都倾向这种方式。规则被称为**生成式**，因为它们生成了语法中的字符串。

上下文无关语法中的每个生成式都有一个**头部**（其名称）和描述其生成内容的**主体**。在纯粹的形式上看，主体只是一系列符号。符号有两种：

- **终结符**是语法字母表中的一个字母。你可以把它想象成一个字面值。在我们定义的语法中，终止符是独立的词素——来自扫描器的词法标记，比如 if 或 1234。这些词素被称为“终止符”，表示“终点”，因为它们不会导致游戏中任何进一步的" 动作"。你只是简单地产生了那一个符号。
- 非终止符是对语法中另一条规则的命名引用。它的意思是“执行那条规则，然后将它产生的任何内容插入这里”。这样，语法就构成了。

还有最后一个细节：你可以有多个同名的规则。当你遇到一个该名字的非终止符时，你可以为它选择任何一条规则，随您喜欢。

为了让这个规则具体化，我们需要一种方式来写下这些生成规则。人们一直试图将语法具体化，可以追溯到 Pini 的 *Ashtadhyayi*，他在几千年前编纂了梵文语法。直到约翰-巴科斯（John Backus）和公司需要一个声明 ALGOL 58 的符号，并提出了巴科斯范式（**BNF**），才有了很大的进展。从那时起，几乎每个人都在使用 BNF 的某种变形，并根据自己的需要进行了调整。

我试图提出一个简单的形式。每个规则都是一个名称，后跟一个箭头（→），后跟一系列符号，最后以分号（;）结尾。终止符是带引号的字符串，非终止符是小写的单词。

以此为基础，下面是一个早餐菜单语法：

```
breakfast  → protein "with" breakfast "on the side" ;
breakfast  → protein ;
breakfast  → bread ;

protein    → crispiness "crispy" "bacon" ;
protein    → "sausage" ;
protein    → cooked "eggs" ;

crispiness → "really" ;
crispiness → "really" crispiness ;

cooked     → "scrambled" ;
cooked     → "poached" ;
cooked     → "fried" ;

bread      → "toast" ;
bread      → "biscuits" ;
bread      → "English muffin" ;
```

我们可以使用这个语法来随机生成早餐。我们来玩一轮，看看它是如何工作的。按照老规矩，游戏从语法中的第一个规则开始，这里是 breakfast。它有三个生成式，我们随机选择第一个。我们得到的字符串是这样的：

```
protein "with" breakfast "on the side"
```

我们需要展开第一个非终止符，protein，所有我们要选择它对应的一个生成式。我们选：

```
protein → cooked "eggs" ;
```

接下来，我们需要 cooked 的生成式，我们选择“poached”。这是一个终止符，我们加上它。现在我们的字符串是这样的：

```
"poached" "eggs" "with" breakfast "on the side"
```

下一个非终止符还是 breakfast，我们开始选择的 breakfast 生成式递归地指向了 breakfast 规则。语法中的递归是一个很好的标志，表明所定义的语言是上下文无关的，而不是正则的。特别是，递归非终止符两边都有生成式的递归，意味着语言不是正则的。

我们可以不断选择 breakfast 的第一个生成式，以做出各种各样的早餐：“bacon with sausage with scrambled eggs with bacon.....”，【存疑，按照规则设置，这里应该不会出现以 bacon 开头的字符串，原文可能有误】但我们不会这样做。这一次我们选择 bread。有三个对应的规则，每个规则只包含一个终止符。我们选“English muffin”。

这样一来，字符串中的每一个非终止符都被展开了，直到最后只包含终止符，我们就剩下：

再加上一些火腿和荷兰酱，你就得到了松饼蛋。

每当我们遇到具有多个作品的规则时，我们都只是随意选择了一个。正是这种灵活性允许用少量的语法规则来编码出组合性更强的字符串集。一个规则可以直接或间接地引用它自己，这就更提高了它的灵活性，让我们可以将无限多的字符串打包到一个有限的语法中。

5.1.2 增强符号

在少量的规则中可以填充无限多的字符串是相当奇妙的，但是我们可以更进一步。我们的符号是可行的，但有点乏味。所以，就像所有优秀的语言设计者一样，我们会在上面撒一些语法糖。除了终止符和非终止符之外，我们还允许在规则的主体中使用一些其他类型的表达式：

- 我们将允许一系列由管道符 (|) 分隔的生成式，避免在每次在添加另一个生成式时重复规则名称。

  ```
  bread → "toast" | "biscuits" | "English muffin" ;
  ```

- 此外，我们允许用括号进行分组，然后在分组中可以用 | 表示从一系列生成式中选择一个。

  ```
  protein → ( "scrambled" | "poached" | "fried" ) "eggs" ;
  ```

- 使用递归来支持符号的重复序列有一定的吸引力，但每次我们要循环的时候，都要创建一个单独的命名子规则，有点繁琐。所以，我们也使用后缀 * 来允许前一个符号或组重复零次或多次。

```
crispiness → "really" "really"* ;
```

- 后缀 + 与此类似，但要求前面的生成式至少出现一次。

```
crispiness → "really"+ ;
```

- 后缀? 表示可选生成式，它之前的生成式可以出现零次或一次，但不能出现多次。

```
breakfast → protein ( "with" breakfast "on the side" )? ;
```

- 有了所有这些语法上的技巧，我们的早餐语法浓缩为：

```
breakfast → protein ( "with" breakfast "on the side" )?
          | bread ;

protein   → "really"+ "crispy" "bacon"
          | "sausage"
          | ( "scrambled" | "poached" | "fried" ) "eggs" ;

bread     → "toast" | "biscuits" | "English muffin" ;
```

我希望还不算太坏。如果你习惯使用 grep 或在你的文本编辑器中使用正则表达式，大多数的标点符号应该是熟悉的。主要区别在于，这里的符号代表整个标记，而不是单个字符。

在本书的其余部分中，我们将使用这种表示法来精确地描述 Lox 的语法。当您使用编程语言时，您会发现上下文无关的语法 (使用此语法或 EBNF 或其他一些符号) 可以帮助您将非正式的语法设计思想具体化。它们也是与其他语言黑客交流语法的方便媒介。

我们为 Lox 定义的规则和生成式也是我们将要实现的树数据结构（用于表示内存中的代码）的指南。在此之前，我们需要为 Lox 编写一个实际的语法，或者至少要有一个足够上手的语法。

5.1.3 Lox 表达式语法

在上一章中，我们一气呵成地完成了 Lox 的全部词汇语法，包括每一个关键词和标点符号。但句法语法的规模更大，如果在我们真正启动并运行解释器之前，就要把整个语法啃完，那就太无聊了。

相反，我们将在接下来的几章中摸索该语言的一个子集。一旦我们可以对这个迷你语言进行表示、解析和解释，那么在之后的章节中将逐步为它添加新的特性，包括新的语法。现在，我们只关心几个表达式：

- **字面量**。数字、字符串、布尔值以及 nil。
- **一元表达式**。前缀！执行逻辑非运算，-对数字求反。
- **二元表达式**。我们已经知道的中缀算术符（+，-，*，/）和逻辑运算符（==，!=，<，<=，>，>=）。
- **括号**。表达式前后的一对 (和)。

这已经为表达式提供了足够的语法，例如：

```
1 - (2 * 3) < 4 == false
```

使用我们的新符号，下面是语法的表示：

```
expression     → literal
               | unary
               | binary
               | grouping ;

literal        → NUMBER | STRING | "true" | "false" | "nil" ;
grouping       → "(" expression ")" ;
unary          → ( "-" | "!" ) expression ;
binary         → expression operator expression ;
operator       → "==" | "!=" | "<" | "<=" | ">" | ">="
               | "+" | "-" | "*" | "/" ;
```

这里有一点额外的元语法。除了与精确词素相匹配的终止符会加引号外，我们还对表示单一词素的终止符进行“大写化”，这些词素的文本表示方式可能会有所不同。NUMBER 是任何数字字面量，STRING 是任何字符串字面量。稍后，我们将对 IDENTIFIER 进行同样的处理。

这个语法实际上是模棱两可的，我们在解析它时就会看到这一点。但现在这已经足够了。

5.2 实现语法树

最后，我们要写一些代码。这个小小的表达式语法就是我们的骨架。由于语法是递归的——请注意 grouping，unary，和 binary 都是指回 expression 的——我们的数据结构将形成一棵树。因为这个结构代表了我们语言的语法，所以叫做**语法树**。

我们的扫描器使用一个单一的 Token 类来表示所有类型的词素。为了区分不同的种类——想想数字 123 和字符串"123"——我们创建了一个简单的 TokenType 枚举。语法树并不是那么同质的。一元表达式只有一个操作数，二元表达式有两个操作数，而字面量则没有。

我们可以将所有这些内容整合到一个包含任意子类列表的 Expression 类中。有些编译器会这么做。但我希望充分利用 Java 的类型系统。所以我们将为表达式定义一个基类。

然后，对于每一种表达式——expression 下的每一个生成式——我们创建一个子类，这个子类有该规则所特有的非终止符字段。这样，如果试图访问一元表达式的第二个操作数，就会得到一个编译错误。

类似这样：

```
package com.craftinginterpreters.lox;

abstract class Expr {
 static class Binary extends Expr {
  Binary(Expr left, Token operator, Expr right) {
   this.left = left;
   this.operator = operator;
   this.right = right;
  }

  final Expr left;
  final Token operator;
  final Expr right;
 }

 // Other expressions...
}
```

Expr 是所有表达式类继承的基类。从 Binary 中可以看到，子类都嵌套在它的内部。这在技术上没有必要，但它允许我们将所有类都塞进一个 Java 文件中。

5.2.1 非面向对象

你会注意到，（表达式类）像 Token 类一样，其中没有任何方法。这是一个很愚蠢的结构，巧妙的类型封装，但仅仅是一包数据。这在 Java 这样的面向对象语言中会有些奇怪，难道类不是应该做一些事情吗？

问题在于这些树类不属于任何单个的领域。树是在解析的时候创建的，难道类中应该有解析对应的方法？或者因为树结构在解释的时候被消费，其中是不是要提供解释相关的方法？树跨越了这些领域之间的边界，这意味着它们实际上不属于任何一方。

事实上，这些类型的存在是为了让解析器和解释器能够进行交流。这就适合于那些只是简单的数据而没有相关行为的类型。这种风格在 Lisp 和 ML 这样的函数式语言中是非常自然的，因为在这些语言中，所有的数据和行为都是分开的，但是在 Java 中感觉很奇怪。

函数式编程的爱好者们现在都跳起来惊呼：“看吧！面向对象的语言不适合作为解释器！”我不会那么过分的。您可能还记得，扫描器本身非常适合面向对象。它包含所有的

可变状态来跟踪其在源代码中的位置、一组定义良好的公共方法和少量的私有辅助方法。

我的感觉是，在面向对象的风格下，解释器的每个阶段或部分都能正常工作。只不过在它们之间流动的数据结构剥离了行为。

5.2.2 语法树的元编程

Java 可以表达无行为的类，但很难说它特别擅长。用 11 行代码在一个对象中填充 3 个字段是相当乏味的，当我们全部完成后，我们将有 21 个这样的类。

我不想浪费你的时间或我的墨水把这些都写下来。真的，每个子类的本质是什么？一个名称和一个字段列表而已。我们是聪明的语言黑客，对吧？我们把它自动化。

与其繁琐地手写每个类的定义、字段声明、构造函数和初始化器，我们一起编写一个脚本来完成任务。它具有每种树类型（名称和字段）的描述，并打印出定义具有该名称和状态的类所需的 Java 代码。

该脚本是一个微型 Java 命令行应用程序，它生成一个名为“Expr.java”的文件：

tool/GenerateAst.java，创建新文件

```
package com.craftinginterpreters.tool;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.Arrays;
import java.util.List;

public class GenerateAst {
 public static void main(String[] args) throws IOException {
  if (args.length != 1) {
   System.err.println("Usage: generate_ast <output directory>");
   System.exit(64);
  }
  String outputDir = args[0];
 }
}
```

注意，这个文件在另一个包中，是.tool 而不是.lox。这个脚本并不是解释器本身的一部分，它是一个工具，我们这种编写解释器的人，通过运行该脚本来生成语法树类。完成后，我们把“Expr.java”与实现中的其它文件进行相同的处理。我们只是自动化了文件的生成方式。

为了生成类，还需要对每种类型及其字段进行一些描述。

tool/GenerateAst.java，在 main() 方法中添加

```
    String outputDir = args[0];
    defineAst(outputDir, "Expr", Arrays.asList(
      "Binary   : Expr left, Token operator, Expr right",
      "Grouping : Expr expression",
      "Literal  : Object value",
      "Unary    : Token operator, Expr right"
    ));
  }
```

为简便起见，我将表达式类型的描述放入了字符串中。每一项都包括类的名称，后跟：和以逗号分隔的字段列表。每个字段都有一个类型和一个名称。

defineAst() 需要做的第一件事是输出基类 Expr。

tool/GenerateAst.java，在 main() 方法后添加：

```
  private static void defineAst(
      String outputDir, String baseName, List<String> types)
      throws IOException {
    String path = outputDir + "/" + baseName + ".java";
    PrintWriter writer = new PrintWriter(path, "UTF-8");

    writer.println("package com.craftinginterpreters.lox;");
    writer.println();
    writer.println("import java.util.List;");
    writer.println();
    writer.println("abstract class " + baseName + " {");

    writer.println("}");
    writer.close();
  }
```

我们调用这个函数时，baseName 是“Expr”，它既是类的名称，也是它输出的文件的名称。我们将它作为参数传递，而不是对名称进行硬编码，因为稍后我们将为语句添加一个单独的类族。

在基类内部，我们定义每个子类。

tool/GenerateAst.java，在 defineAst() 类中添加：

```
    writer.println("abstract class " + baseName + " {");
    // The AST classes.
    for (String type : types) {
      String className = type.split(":")[0].trim();
      String fields = type.split(":")[1].trim();
      defineType(writer, baseName, className, fields);
    }
    writer.println("}");
```

这段代码依次调用：

tool/GenerateAst.java，在 defineAst() 后面添加：

```
private static void defineType(
    PrintWriter writer, String baseName,
    String className, String fieldList) {
  writer.println("  static class " + className + " extends " +
      baseName + " {");

  // Constructor.
  writer.println("    " + className + "(" + fieldList + ") {");

  // Store parameters in fields.
  String[] fields = fieldList.split(", ");
  for (String field : fields) {
    String name = field.split(" ")[1];
    writer.println("      this." + name + " = " + name + ";");
  }

  writer.println("    }");

  // Fields.
  writer.println();
  for (String field : fields) {
    writer.println("    final " + field + ";");
  }

  writer.println("  }");
}
```

好了。所有的 Java 模板都完成了。它在类体中声明了每个字段。它为类定义了一个构造函数，为每个字段提供参数，并在类体中对其初始化。

现在编译并运行这个 Java 程序，它会生成一个新的“.java”文件，其中包含几十行代码。那份文件还会变得更长。

5.3 处理树形结构

先想象一下吧。尽管我们还没有到那一步，但请考虑一下解释器将如何处理语法树。Lox 中的每种表达式在运行时的行为都不一样。这意味着解释器需要选择不同的代码块来处理每种表达式类型。对于词法标记，我们可以简单地根据 TokenType 进行转换。但

是我们并没有为语法树设置一个“type”枚举，只是为每个语法树单独设置一个 Java 类。

我们可以编写一长串类型测试：

```
if (expr instanceof Expr.Binary) {
  // ...
} else if (expr instanceof Expr.Grouping) {
  // ...
} else // ...
```

但所有这些顺序类型测试都很慢。类型名称按字母顺序排列在后面的表达式，执行起来会花费更多的时间，因为在找到正确的类型之前，它们会遇到更多的 if 情况。这不是我认为的优雅解决方案。

我们有一个类族，我们需要将一组行为与每个类关联起来。在 Java 这样的面向对象语言中，最自然的解决方案是将这些行为放入类本身的方法中。我们可以在 Expr 上添加一个抽象的 interpret() 方法，然后每个子类都要实现这个方法来解释自己。

这对于小型项目来说还行，但它的扩展性很差。就像我之前提到的，这些树类跨越了几个领域。至少，解析器和解释器都会对它们进行干扰。稍后您将看到，我们需要对它们进行名称解析。如果我们的语言是静态类型的，我们还需要做类型检查。

如果我们为每一个操作的表达式类中添加实例方法，就会将一堆不同的领域混在一起。这违反了关注点分离原则，并会产生难以维护的代码。

5.3.1 表达式问题

这个问题比起初看起来更基础。我们有一些类型，和一些高级操作，比如“解释”。对于每一对类型和操作，我们都需要一个特定的实现。画一个表：

行是类型，列是操作。每个单元格表示在该类型上实现该操作的唯一代码段。

像 Java 这样的面向对象的语言，假定一行中的所有代码都自然地挂在一起。它认为你对一个类型所做的所有事情都可能是相互关联的，而使用这类语言可以很容易将它们一起定义为同一个类里面的方法。

这种情况下，向表中加入新行来扩展列表是很容易的，简单地定义一个新类即可，不需要修改现有的代码。但是，想象一下，如果你要添加一个新操作（新的一行）。在 Java 中，这意味着要拆开已有的那些类并向其中添加方法。

ML 家族中的函数式范型反过来了。在这些语言中，没有带方法的类，类型和函数是完全独立的。要为许多不同类型实现一个操作，只需定义一个函数。在该函数体中，您可以使用 * 模式匹配 *（某种基于类型的 switch 操作）在同一个函数中实现每个类型对应的操作。

这使得添加新操作非常简单——只需定义另一个与所有类型模式匹配的的函数即可。

但是，反过来说，添加新类型是困难的。您必须回头向已有函数中的所有模式匹配添加一个新的 case。

每种风格都有一定的“纹路”。这就是范式名称的字面意思——面向对象的语言希望你按照类型的行来组织你的代码。而函数式语言则鼓励你把每一列的代码都归纳为一个函数。

一群聪明的语言迷注意到，这两种风格都不容易向表格中添加行和列。他们称这个困难为“表达式问题”。就像我们现在一样，他们是在试图找出在编译器中建模表达式语法树节点的最佳方法时，第一次遇到了该问题。

人们已经抛出了各种各样的语言特性、设计模式和编程技巧，试图解决这个问题，但还没有一种完美的语言能够解决它。与此同时，我们所能做的就是尽量选择一种与我们正在编写的程序的自然架构相匹配的语言。

面向对象在我们的解释器的许多部分都可以正常工作，但是这些树类与 Java 的本质背道而驰。幸运的是，我们可以采用一种设计模式来解决这个问题。

5.3.2 访问者模式

访问者模式是所有设计模式中最容易被误解的模式，当您回顾过去几十年的软件架构泛滥状况时，会发现确实如此。

问题出在术语上。这个模式不是关于“visiting（访问）”，它的“accept”方法也没有让人产生任何有用的想象。许多人认为这种模式与遍历树有关，但事实并非如此。我们确实要在一组树结构的类上使用它，但这只是一个巧合。如您所见，该模式在单个对象上也可以正常使用。

访问者模式实际上近似于 OOP 语言中的函数式。它让我们可以很容易地向表中添加新的列。我们可以在一个地方定义针对一组类型的新操作的所有行为，而不必触及类型本身。这与我们解决计算机科学中几乎所有问题的方式相同：添加中间层。

在将其应用到自动生成的 Expr 类之前，让我们先看一个更简单的例子。比方说我们有两种点心：Beignet(卷饼) 和 Cruller(油酥卷)。

```
abstract class Pastry {
}

class Beignet extends Pastry {
}

class Cruller extends Pastry {
}
```

我们希望能够定义新的糕点操作（烹饪，食用，装饰等），而不必每次都向每个类添加新方法。我们是这样做的。首先，我们定义一个单独的接口。

```
interface PastryVisitor {
 void visitBeignet(Beignet beignet);
 void visitCruller(Cruller cruller);
}
```

可以对糕点执行的每个操作都是实现该接口的新类。它对每种类型的糕点都有具体的方法。这样一来，针对两种类型的操作代码都紧密地嵌套在一个类中。

给定一个糕点，我们如何根据其类型将其路由到访问者的正确方法？多态性拯救了我们！我们在 Pastry 中添加这个方法：

```
abstract class Pastry {
 abstract void accept(PastryVisitor visitor);
}
```

每个子类都需要实现该方法：

```
class Beignet extends Pastry {
 @Override
 void accept(PastryVisitor visitor) {
  visitor.visitBeignet(this);
 }
}
```

以及：

```
class Cruller extends Pastry {
 @Override
 void accept(PastryVisitor visitor) {
  visitor.visitCruller(this);
 }
}
```

要对糕点执行一个操作，我们就调用它的 accept() 方法，并将我们要执行的操作 vistor 作为参数传入该方法。pastry 类——特定子类对 accept() 的重写实现——会反过来，在 visitor 上调用合适的 visit 方法，并将自身作为参数传入。

这就是这个技巧的核心所在。它让我们可以在 *pastry* 类上使用多态派遣，在 *visitor* 类上选择合适的方法。对应在表格中，每个 pastry 类都是一行，但如果你看一个 visitor 的所有方法，它们就会形成一列。

我们为每个类添加了一个 accept() 方法，我们可以根据需要将其用于任意数量的访问者，而无需再次修改 *pastry* 类。这是一个聪明的模式。

5.3.3 表达式访问者

好的，让我们将它编入表达式类中。我们还要对这个模式进行一下完善。在糕点的例子中，visit 和 accept() 方法没有返回任何东西。在实践中，访问者通常希望定义能够产生值的操作。但 accept() 应该具有什么返回类型呢？我们不能假设每个访问者类都想产生相同的类型，所以我们将使用泛型来让每个实现类自行填充一个返回类型。

首先，我们定义访问者接口。同样，我们把它嵌套在基类中，以便将所有的内容都放在一个文件中。

tool/GenerateAst.java，在 defineAst() 方法中添加：

```
    writer.println("abstract class " + baseName + " {");
    defineVisitor(writer, baseName, types);
    // The AST classes.
```

这个函数会生成 visitor 接口。

tool/GenerateAst.java，在 defineAst() 方法后添加：

```
  private static void defineVisitor(
      PrintWriter writer, String baseName, List<String> types) {
    writer.println("  interface Visitor<R> {");

    for (String type : types) {
      String typeName = type.split(":")[0].trim();
      writer.println("    R visit" + typeName + baseName + "(" +
          typeName + " " + baseName.toLowerCase() + ");");
    }

    writer.println("  }");
  }
```

在这里，我们遍历所有的子类，并为每个子类声明一个 visit 方法。当我们以后定义新的表达式类型时，会自动包含这些内容。

在基类中，定义抽象 accept() 方法。

tool/GenerateAst.java，在 defineAst() 方法中添加：

```
      defineType(writer, baseName, className, fields);
    }
    // The base accept() method.
    writer.println();
    writer.println("  abstract <R> R accept(Visitor<R> visitor);");
    writer.println("}");
```

最后，每个子类都实现该方法，并调用其类型对应的 visit 方法。

tool/GenerateAst.java，在 defineType() 方法中添加：

```
    writer.println("  }");
    // Visitor pattern.
    writer.println();
    writer.println("  @Override");
    writer.println("  <R> R accept(Visitor<R> visitor) {");
    writer.println("    return visitor.visit" +
        className + baseName + "(this);");
    writer.println("  }");
    // Fields.
```

这下好了。现在我们可以在表达式上定义操作，而且无需对类或生成器脚本进行修改。编译并运行这个生成器脚本，输出一个更新后的 “Expr.java” 文件。该文件中包含一个生成的 Visitor 接口和一组使用该接口支持 Visitor 模式的表达式节点类。

在结束这一章的闲聊之前，我们先实现一下这个 Visitor 接口，看看这个模式的运行情况。

5.4 一个（不是很）漂亮的打印器

当我们调试解析器和解释器时，查看解析后的语法树并确保其与期望的结构一致通常是很有用的。我们可以在调试器中进行检查，但那可能有点难。

相反，我们需要一些代码，在给定语法树的情况下，生成一个明确的字符串表示。将语法树转换为字符串是解析器的逆向操作，当我们的目标是产生一个在源语言中语法有效的文本字符串时，通常被称为" 漂亮打印"。

这不是我们的目标。我们希望字符串非常明确地显示树的嵌套结构。如果我们要调试的是操作符的优先级是否处理正确，那么返回 1 + 2 * 3 的打印器并没有什么用，我们想知道 + 或 * 是否在语法树的顶部。

因此，我们生成的字符串表示形式不是 Lox 语法。相反，它看起来很像 Lisp。每个表达式都被显式地括起来，并且它的所有子表达式和词法标记都包含在其中。

给定一个语法树，如：

输出结果为：

```
(* (- 123) (group 45.67))
```

不是很 “漂亮”，但是它确实明确地展示了嵌套和分组。为了实现这一点，我们定义了一个新类。

lox/AstPrinter.java，创建新文件：

```
package com.craftinginterpreters.lox;

class AstPrinter implements Expr.Visitor<String> {
 String print(Expr expr) {
  return expr.accept(this);
 }
}
```

如你所见，它实现了 visitor 接口。这意味着我们需要为我们目前拥有的每一种表达式类型提供 visit 方法。

lox/AstPrinter.java，在 print() 方法后添加：

```
 return expr.accept(this);
 }

 @Override
 public String visitBinaryExpr(Expr.Binary expr) {
  return parenthesize(expr.operator.lexeme,
                      expr.left, expr.right);
 }

 @Override
 public String visitGroupingExpr(Expr.Grouping expr) {
  return parenthesize("group", expr.expression);
 }

 @Override
 public String visitLiteralExpr(Expr.Literal expr) {
  if (expr.value == null) return "nil";
  return expr.value.toString();
 }

 @Override
 public String visitUnaryExpr(Expr.Unary expr) {
  return parenthesize(expr.operator.lexeme, expr.right);
 }
}
```

字面量表达式很简单——它们将值转换为一个字符串，并通过一个小检查用 Java 中的 null 代替 Lox 中的 nil。其他表达式有子表达式，所以它们要使用 parenthesize() 这个辅助方法：

lox/AstPrinter.java，在 visitUnaryExpr() 方法后添加：

```
  private String parenthesize(String name, Expr... exprs) {
    StringBuilder builder = new StringBuilder();

    builder.append("(").append(name);
    for (Expr expr : exprs) {
      builder.append(" ");
      builder.append(expr.accept(this));
    }
    builder.append(")");

    return builder.toString();
  }
```

它接收一个名称和一组子表达式作为参数，将它们全部包装在圆括号中，并生成一个如下的字符串：

```
(+ 1 2)
```

请注意，它在每个子表达式上调用 accept() 并将自身传递进去。这是递归步骤，可让我们打印整棵树。

我们还没有解析器，所以很难看到它的实际应用。现在，我们先使用一个 main() 方法来手动实例化一个树并打印它。

lox/AstPrinter.java，在 parenthesize() 方法后添加：

```
  public static void main(String[] args) {
    Expr expression = new Expr.Binary(
        new Expr.Unary(
            new Token(TokenType.MINUS, "-", null, 1),
            new Expr.Literal(123)),
        new Token(TokenType.STAR, "*", null, 1),
        new Expr.Grouping(
            new Expr.Literal(45.67)));

    System.out.println(new AstPrinter().print(expression));
  }
```

如果我们都做对了，它就会打印：

```
(* (- 123) (group 45.67))
```

您可以继续删除这个方法，我们后面不再需要它了。另外，当我们添加新的语法树类型时，我不会在 AstPrinter 中展示它们对应的 visit 方法。如果你想这样做 (并且希望 Java 编译器不会报错)，那么你可以自行添加这些方法。在下一章，当我们开始将 Lox 代码解析为语法树时，它将会派上用场。或者，如果你不想维护 AstPrinter，可以随意删除它。我们不再需要它了。

5.5 挑战

1. 之前我说过，我们在语法元语法中添加的 |、*、+ 等形式只是语法糖。以这个语法为例:

```
expr → expr ( "(" ( expr ( "," expr )* )? ")" | "." IDENTIFIER )+
     | IDENTIFIER
     | NUMBER
```

   生成一个与同一语言相匹配的语法，但不要使用任何语法糖。

   附加题：这一点语法表示了什么样的表达式?

2. Visitor 模式让你可以在面向对象的语言中模仿函数式。为函数式语言设计一个互补的模式，该模式让你可以将一个类型上的所有操作捆绑在一起，并轻松扩展新的类型。

   (SML 或 Haskell 是这个练习的理想选择，但 Scheme 或其它 Lisp 方言也可以。)

3. 在逆波兰表达式 (RPN) 中，算术运算符的操作数都放在运算符之前，所以 1 + 2 变成了 1 2 +。计算时从左到右进行，操作数被压入隐式栈。算术运算符弹出前两个数字，执行运算，并将结果推入栈中。因此,

```
(1 + 2) * (4 - 3)
```

   在 RPN 中变为了

```
1 2 + 4 3 - *
```

   为我们的语法树类定义一个 Vistor 类，该类接受一个表达式，将其转换为 RPN，并返回结果字符串。

第六章 解析表达式

> 语法，它甚至知道如何控制国王。
>
> ——莫里哀

本章是本书的第一个重要里程碑。我们中的许多人都曾将正则表达式和字符串操作的糅合在一起，以便从一堆文本中提取一些信息。这些代码可能充满了错误，而且很难维护。编写一个真正的解析器——具有良好的错误处理、一致的内部结构和能够健壮地分析复杂语法的能力——被认为是一种罕见的、令人印象深刻的技能。在这一章中，你将获得这种技能。

这比想象中要简单，部分是因为我们在上一章中提前完成了很多困难的工作。你已经对形式化语法了如指掌，也熟悉了语法树，而且我们有一些 Java 类来表示它们。唯一剩下的部分是解析——将一个标记序列转换成这些语法树中的一个。

一些 CS 教科书在解析器上大做文章。在 60 年代，计算机科学家——他们理所当然地厌倦了用汇编语言编程——开始设计更复杂的、对人类友好的语言，比如 Fortran 和 ALGOL。唉，对于当时原始的计算机来说，这些语言对机器并不友好。

这些先驱们设计了一些语言，说实话，他们甚至不知道如何编写编译器。然后他们做了开创性的工作，发明了解析和编译技术，可以在那些老旧、小型的机器上处理这些新的、大型的语言。

经典的编译书读起来就像是对这些英雄和他们的工具的吹捧传记。《编译器: 原理、技术和工具》（*Compilers: Principles, Techniques, and Tools*）的封面上有一条标记着“编译器设计复杂性”的龙，被一个手持剑和盾的骑士杀死，剑和盾上标记着“LALR 解析器生成器”和“语法制导翻译”。他们在过分吹捧。

稍微的自我祝贺是当之无愧的，但事实是，你不需要知道其中的大部分知识，就可以为现代机器制作出高质量的解析器。一如既往，我鼓励你先扩大学习范围，以后再慢慢接受它，但这本书省略了奖杯箱。

6.1 歧义与解析游戏

在上一章中，我说过你可以像“玩”游戏一样使用上下文无关的语法来 * 生成 * 字符串。解析器则以相反的方式玩游戏。给定一个字符串 (一系列语法标记)，我们将这些标记映射到语法中的终止符，以确定哪些规则可能生成该字符串。

" 可能产生" 这部分很有意思。我们完全有可能创建一个 * 模棱两可 * 的语法，在这个语法中，不同的生成式可能会得到同一个字符串。当你使用该语法来 * 生成 * 字符串时，这一点不太重要。一旦你有了字符串，谁还会在乎你是怎么得到它的呢?

但是在解析时，歧义意味着解析器可能会误解用户的代码。当我们进行解析时，我们不仅要确定字符串是不是有效的 Lox 代码，还要记录哪些规则与代码的哪些部分相匹

配，以便我们知道每个标记属于语言的哪一部分。下面是我们在上一章整理的 Lox 表达式语法：

```
expression     literal
               unary
               binary
               grouping ;

literal        NUMBER | STRING | "true" | "false" | "nil" ;
grouping       "(" expression ")" ;
unary          ( "-" | "!" ) expression ;
binary         expression operator expression ;
operator       "==" | "!=" | "<" | "<=" | ">" | ">="
               | "+"  | "-"  | "*" | "/" ;
```

下面是一个满足语法的有效字符串：

但是，有两种方式可以生成该字符串。其一是：

1. 从 ‘expression‘开始，选择 ‘binary‘。2. 对于左边的 ‘expression‘，选择 ‘NUMBER‘，并且使用 ‘6‘。3. 对于操作符，选择 ‘/‘。4. 对于右边的 ‘expression‘，再次选择 ‘binary‘。5. 在内层的 ‘binary‘ 表达式中，选择 ‘3-1‘。

其二是：

1. 从 ‘expression‘开始，选择 ‘binary‘。2. 对于左边的 ‘expression‘，再次选择 ‘binary‘。3. 在内层的 ‘binary‘ 表达式中，选择 ‘6/3‘。4. 返回外层的 ‘binary‘ ，对于操作符，选择 ‘-‘。5. 对于右边的 ‘expression‘，选择 ‘NUMBER‘，并且使用 ‘1‘。

它们产生相同的字符串，但对应的是不同的 * 语法树 *：

换句话说，这个语法可以将该表达式看作是 ‘(6 / 3) - 1‘或 ‘6 / (3 - 1)‘。‘binary‘ 规则运行操作数以任意方式嵌套，这反过来又会影响解析数的计算结果。自从黑板被发明以来，数学家们解决这种模糊性的方法就是定义优先级和结合性规则。

- ** 优先级 ** 决定了在一个包含不同运算符的混合表达式中，哪个运算符先被执行。优先级规则告诉我们，在上面的例子中，我们在 ‘-‘之前先计算 ‘/‘。优先级较高的运算符在优先级较低的运算符之前计算。同样，优先级较高的运算符被称为" 更严格的绑定"。

- ** 结合性 ** 决定在一系列相同操作符中先计算哪个操作符。如果一个操作符是 ** 左结合 ** 的 (可以认为是 “从左到右”) 时，左边的操作符在右边的操作符之前计算。因为 ‘-‘是左结合的，下面的表达式：

```
5 - 3 - 1
```

等价于：

```
(5 - 3) - 1
```

另一方面，赋值是 ** 右结合 ** 的。如：

```
a = b = c
```

等价于：

```
a = (b = c)
```

如果没有明确定义的优先级和结合性，使用多个运算符的表达式可能就会变得有歧义——它可以被解析为不同的语法树，而这些语法树又可能会计算出不同的结果。我们在 Lox 中会解决这个问题，使用与 C 语言相同的优先级规则，从低到高分别是：

| Name | Operators | Associates | | ———— | ———— | ———- | | Equality 等于 | ‘==‘ ‘!=‘ | Left 左结合 | | Comparison 比较 | ‘>‘ ‘>=‘ ‘<‘ ‘<=‘ | Left 左结合 | | Term 加减运算 | ‘-‘ ‘+‘ | Left 左结合 | | Factor 乘除运算 | ‘/‘ ‘*‘ | Left 左结合 | | Unary 一元运算符 | ‘¡ ‘-‘ | Right 右结合 |

现在，该语法将所有表达式类型都添加到一个 ‘expression‘规则中。这条规则同样作用于操作数中的非终止符，这使得语法中可以接受任何类型的表达式作为子表达式，而不管优先级规则是否允许。

我们通过对语法进行分层来解决这个问题。我们为每个优先级定义一个单独的规则。

```
expression     | ...
equality       | ...
comparison     | ...
term           | ...
factor         | ...
unary          | ...
primary        | ...
```

此处的每个规则仅匹配其当前优先级或更高优先级的表达式。例如，‘unary‘ 匹配一元表达式（如 ‘!negated‘）或主表达式（如 ‘1234‘）。‘term‘可以匹配 ‘1 + 2‘，但也可以匹配 ‘3 * 4 /5‘。最后的 ‘primary‘ 规则涵盖优先级最高的形式——字面量和括号表达式。

我们只需要填写每条规则的生成式。我们先从简单的开始。顶级的 ‘expression‘ 规则可以匹配任何优先级的表达式。由于 ‘equality‘的优先级最低，只要我们匹配了它，就涵盖了一切。

```
expression → equality
```

> Over at the other end of the precedence table, a primary expression contains all the literals and grouping expressions.

在优先级表的另一端，‘primary‘表达式包括所有的字面量和分组表达式。

```
primary        | NUMBER | STRING | "true" | "false" | "nil"
               | "(" expression ")" ;
```

一元表达式以一元操作符开头，后跟操作数。因为一元操作符可以嵌套——‘!!true‘虽奇怪也是可用的表达式——这个操作数本身可以是一个一元表达式。递归规则可以很好地解决这个问题。

```
unary          | ( "!" | "-" ) unary ;
```

但是这条规则有一个问题，它永远不会终止。

请记住，每个规则都需要匹配该优先级或更高优先级的表达式，因此我们还需要使其与主表达式匹配。

```
unary          | ( "!" | "-" ) unary
               | primary ;
```

这样就可以了。

剩下的规则就是二元运算符。我们先从乘法和除法的规则开始。下面是第一次尝试：

```
factor         | factor ( "/" | "*" ) unary
               | unary ;
```

该规则递归匹配左操作数，这样一来，就可以匹配一系列乘法和除法表达式，例如‘1 * 2 / 3‘。将递归生成式放在左侧并将 ‘unary‘ 放在右侧，可以使该规则具有左关联性和明确性。

所有这些都是正确的，但规则主体中的第一个符号与规则头部相同意味着这个生成式是 ** 左递归 ** 的。一些解析技术，包括我们将要使用的解析技术，在处理左递归时会遇到问题。(其他地方的递归，比如在 ‘unary‘中，以及在 ‘primary‘分组中的间接递归都不是问题。)

你可以定义很多符合同一种语言的语法。如何对某一特定语言进行建模，一部分是品味问题，一部分是实用主义问题。这个规则是正确的，但对于我们后续的解析来说它并不是最优的。我们将使用不同的规则来代替左递归规则。

```
factor         | unary ( ( "/" | "*" ) unary )* ;
```

我们将因子表达式定义为乘法和除法的扁平 * 序列 *。这与前面的规则语法相同，但更好地反映了我们将编写的解析 Lox 的代码。我们对其它二进制运算符的优先级使用相同的结构，从而得到下面这个完整的表达式语法：

```
expression     equality ;
equality       comparison ( ( "!=" | "==" ) comparison )* ;
comparison     term ( ( ">" | ">=" | "<" | "<=" ) term )* ;
term           factor ( ( "-" | "+" ) factor )* ;
factor         unary ( ( "/" | "*" ) unary )* ;
unary          ( "!" | "-" ) unary
               primary ;
primary        NUMBER STRING | "true" | "false" | "nil"
               | "(" expression ")" ;
```

这个语法比我们以前的那个更复杂，但反过来我们也消除了前一个语法定义中的歧义。这正是我们制作解析器时所需要的。

6.2 递归下降分析

现在有一大堆解析技术，它们的名字大多是"L " 和"R " 的组合——LL(k)、LR(1)、LALR——还有更多的异类，比如解析器组合子、Earley parsers、分流码算法和 packrat 解析。对于我们的第一个解释器来说，一种技术已经足够了：** 递归下降 **。

递归下降是构建解析器最简单的方法，不需要使用复杂的解析器生成工具，如 Yacc、Bison 或 ANTLR。你只需要直接手写代码。但是不要被它的简单性所欺骗，递归下降解析器速度快、健壮，并且可以支持复杂的错误处理。事实上，GCC、V8 (Chrome 中的 JavaScript VM)、Roslyn(用 c# 编写的 c# 编译器) 和许多其他重量级产品语言实现都使用了递归下降技术。它很好用。

递归下降被认为是一种 ** 自顶向下解析器 **，因为它从最顶部或最外层的语法规则 (这里是 ‘expression‘) 开始，一直向下进入嵌套子表达式，最后到达语法树的叶子。这与 LR 等自下而上的解析器形成鲜明对比，后者从初级表达式 (primary) 开始，将其组成越来越大的语法块。

递归下降解析器是一种将语法规则直接翻译成命令式代码的文本翻译器。每个规则都会变成一个函数，规则主体翻译成代码大致是这样的：

| Grammar notation | Code representation | | ————————- | ———————————————————— | | Terminal | Code to match and consume a token 匹配并消费一个语法标记 | | Nonterminal | Call to that rule’ s function 调用规则对应的函数 | | ‘|‘ | ‘if‘ or ‘switch‘ statement if 或 switch 语句 | | ‘*‘ or ‘+‘ | ‘while‘ or ‘for‘ loop while 或 for 循环 | | ‘¿ | ‘if‘ statement if 语句 |

下降被 “递归” 修饰是因为，如果一个规则引用自身（直接或间接）就会变为递归的函数调用。

6.2.1 Parser 类

每个语法规则都成为新类中的一个方法:

<u>lox/Parser.java，创建新文件：</u>

```
package com.craftinginterpreters.lox;

import java.util.List;

import static com.craftinginterpreters.lox.TokenType.*;

class Parser {
 private final List<Token> tokens;
 private int current = 0;

 Parser(List<Token> tokens) {
  this.tokens = tokens;
 }
}
```

与扫描器一样，解析器也是消费一个扁平的输入序列，这是这次我们要读取的是语法标记而不是字符。我们会保存标记列表并使用 ‘current‘指向待解析的下一个标记。

我们现在要直接执行表达式语法，并将每一条规则翻译为 Java 代码。第一条规则 ‘expression‘，简单地展开为 ‘equality‘规则，所以很直接：

<u>*lox/Parser.java，在 Parser() 方法添加：*</u>

```
 private Expr expression() {
  return equality();
 }
```

每个解析语法规则的方法都会生成该规则对应的语法树，并将其返回给调用者。当规则主体中包含一个非终止符——对另一条规则的引用时，我们就会调用另一条规则对应的方法。

等式规则有一点复杂：

```
equality       comparison ( ( "!=" | "==" ) comparison )* ;
```

在 Java 中，这会变成：

<u>lox/Parser.java，在 expression() 后面添加：</u>

```
private Expr equality() {
 Expr expr = comparison();

 while (match(BANG_EQUAL, EQUAL_EQUAL)) {
  Token operator = previous();
  Expr right = comparison();
  expr = new Expr.Binary(expr, operator, right);
 }

 return expr;
}
```

让我们一步步来。规则体中的第一个 ‘comparison‘非终止符变成了方法中对 ‘comparison()‘的第一次调用。我们获取结果并将其保存在一个局部变量中。

然后，规则中的 ‘(...)*‘循环映射为一个 ‘while‘循环。我们需要知道何时退出这个循环。可以看到，在规则体中，我们必须先找到一个 ‘ !=‘ 或 ‘ == ‘标记。因此，如果我们 * 没有 * 看到其中任一标记，我们必须结束相等 (不相等) 运算符的序列。我们使用一个方便的 ‘match()‘方法来执行这个检查。

<u>lox/Parser.java，在 equality() 方法后添加：</u>

```
private boolean match(TokenType... types) {
 for (TokenType type : types) {
  if (check(type)) {
   advance();
   return true;
  }
 }

 return false;
}
```

这个检查会判断当前的标记是否属于给定的类型之一。如果是，则消费该标记并返回 ‘true‘；否则，就返回 ‘false‘并保留当前标记。‘match()‘方法是由两个更基本的操作来定义的。

如果当前标记属于给定类型，则 ‘check()‘方法返回 ‘true‘。与 ‘match()‘不同的是，它从不消费标记，只是读取。

<u>lox/Parser.java，在 match() 方法后添加：</u>

```
  private boolean check(TokenType type) {
   if (isAtEnd()) return false;
   return peek().type == type;
  }
```

‘advance()‘方法会消费当前的标记并返回它，类似于扫描器中对应方法处理字符的方式。

<u>lox/Parser.java，在 check() 方法后添加：</u>

```
  private Token advance() {
   if (!isAtEnd()) current++;
   return previous();
  }
```

这些方法最后都归结于几个基本操作。

<u>lox/Parser.java，在 advance() 后添加：</u>

```
  private boolean isAtEnd() {
   return peek().type == EOF;
  }

  private Token peek() {
   return tokens.get(current);
  }

  private Token previous() {
   return tokens.get(current - 1);
  }
```

‘isAtEnd()‘检查我们是否处理完了待解析的标记。‘peek()‘方法返回我们还未消费的当前标记，而 ‘previous()‘会返回最近消费的标记。后者让我们更容易使用 ‘match()‘，然后访问刚刚匹配的标记。

这就是我们需要的大部分解析基本工具。我们说到哪里了？对，如果我们在 ‘equality()‘的 ‘while‘循环中，也就能知道我们已经找到了一个 ‘!=‘或 ‘==‘操作符，并且一定是在解析一个等式表达式。

我们获取到匹配的操作符标记，这样就可以知道我们要处理哪一类等式表达式。之后，我们再次调用 ‘comparison()‘解析右边的操作数。我们将操作符和它的两个操作数组合成一个新的 ‘Expr.Binary‘语法树节点，然后开始循环。对于每一次迭代，我们都将结果表达式存储在同一个 ‘expr‘局部变量中。在对等式表达式序列进行压缩时，会创建一个由二元操作符节点组成的左结合嵌套树。

一旦解析器遇到一个不是等式操作符的标记，就会退出循环。最后，它会返回对应

的表达式。请注意，如果解析器从未遇到过等式操作符，它就永远不会进入循环。在这种情况下，‘equality()‘方法有效地调用并返回 ‘comparison()‘。这样一来，这个方法就会匹配一个等式运算符或 * 任何更高优先级的表达式 *。

继续看下一个规则。

```
comparison     term ( ( ">" | ">=" | "<" | "<=" ) term )* ;
```

翻译成 Java：

<u>lox/Parser.java，在 equality() 方法后添加：</u>

```
  private Expr comparison() {
    Expr expr = term();

    while (match(GREATER, GREATER_EQUAL, LESS, LESS_EQUAL)) {
      Token operator = previous();
      Expr right = term();
      expr = new Expr.Binary(expr, operator, right);
    }

    return expr;
  }
```

语法规则与 ‘equality‘几乎完全相同，相应的代码也是如此。唯一的区别是匹配的操作符的标记类型，而且现在获取操作数时调用的方法是 ‘term()‘而不是 ‘comparison()‘。其余两个二元操作符规则遵循相同的模式。

按照优先级顺序，先做加减法：

<u>lox/Parser.java，在 comparison() 方法后添加：</u>

```
  private Expr term() {
    Expr expr = factor();

    while (match(MINUS, PLUS)) {
      Token operator = previous();
      Expr right = factor();
      expr = new Expr.Binary(expr, operator, right);
    }

    return expr;
  }
```

最后，是乘除法：

<u>lox/Parser.java，在 term() 方法后面添加：</u>

```
  private Expr factor() {
    Expr expr = unary();

    while (match(SLASH, STAR)) {
      Token operator = previous();
      Expr right = unary();
      expr = new Expr.Binary(expr, operator, right);
    }

    return expr;
  }
```

这就是所有的二进制运算符，已经按照正确的优先级和结合性进行了解析。接下来，按照优先级层级，我们要处理一元运算符了。

```
unary          ( "!" | "-" ) unary
               | primary ;
```

该规则对应的代码有些不同。

<u>lox/Parser.java，在 factor() 方法后添加：</u>

```
  private Expr unary() {
    if (match(BANG, MINUS)) {
      Token operator = previous();
      Expr right = unary();
      return new Expr.Unary(operator, right);
    }

    return primary();
  }
```

同样的，我们先检查当前的标记以确认要如何进行解析。如果是‘!‘或‘-‘，我们一定有一个一元表达式。在这种情况下，我们获取令牌，然后递归调用‘unary()‘来解析操作数。将所有这些都包装到一元表达式语法树中，我们就完成了。

否则，我们就达到了最高级别的优先级，即基本表达式。

```
primary        NUMBER STRING "true" | "false" | "nil"
               | "(" expression ")" ;
```

该规则中大部分都是终止符，可以直接进行解析。

<u>lox/Parser.java，在 unary() 方法后添加：</u>

```
  private Expr primary() {
    if (match(FALSE)) return new Expr.Literal(false);
    if (match(TRUE)) return new Expr.Literal(true);
    if (match(NIL)) return new Expr.Literal(null);

    if (match(NUMBER, STRING)) {
      return new Expr.Literal(previous().literal);
    }

    if (match(LEFT_PAREN)) {
      Expr expr = expression();
      consume(RIGHT_PAREN, "Expect ')' after expression.");
      return new Expr.Grouping(expr);
    }
  }
```

有趣的一点是处理括号的分支。当我们匹配了一个开头‘(‘并解析了里面的表达式后，我们必须找到一个‘)‘标记。如果没有找到，那就是一个错误。

6.3 语法错误

解析器实际上有两项工作：

1. > Given a valid sequence of tokens, produce a corresponding syntax tree.

给定一个有效的标记序列，生成相应的语法树。

2. > Given an *invalid* sequence of tokens, detect any errors and tell the user about their mistakes.

给定一个 * 无效 * 的令牌序列，检测错误并告知用户。

不要低估第二项工作的重要性！在现代的 IDE 和编辑器中，为了语法高亮显示和支持自动补齐等功能，当用户还在编辑代码时，解析器就会不断地重新解析代码。这也意味着解析器 * 总是 * 会遇到不完整的、半错误状态的代码。

当用户没有意识到语法错误时，解析器要帮助引导他们回到正确的道路上。它报告错误的方式是你的语言的用户界面中很大的一部分。良好的语法错误处理是很难的。根据定义，代码并不是处于良好定义的状态，所以没有可靠的方法能够知道用户 * 想要 * 写什么。解析器无法读懂你的思想。

当解析器遇到语法错误时，有几个硬性要求。解析器必须能够：

- > **Detect and report the error.** If it doesn’t detect the error and passes the resulting malformed syntax tree on to the interpreter, all manner of horrors may be summoned.

** 检测并报告错误。** 如果它没有检测到错误，并将由此产生的畸形语法树传递给解释器，就会出现各种可怕的情况。

- > **Avoid crashing or hanging.** Syntax errors are a fact of life, and language tools have to be robust in the face of them. Segfaulting or getting stuck in an infinite loop isn't allowed. While the source may not be valid *code*, it's still a valid *input to the parser* because users use the parser to learn what syntax is allowed.

** 避免崩溃或挂起。** 语法错误是生活中不可避免的事实，面对语法错误，语言工具必须非常健壮。段错误或陷入无限循环是不允许的。虽然源代码可能不是有效的 * 代码 *，但它仍然是 * 解析器的有效输入 *，因为用户使用解析器来了解什么是允许的语法。

> Those are the table stakes if you want to get in the parser game at all, but you really want to raise the ante beyond that. A decent parser should:

如果你想参与到解析器的游戏中来，这些就是桌面的筹码，但你真的想提高赌注，除了这些。一个像样的解析器还应该：

- > **Be fast.** Computers are thousands of times faster than they were when parser technology was first invented. The days of needing to optimize your parser so that it could get through an entire source file during a coffee break are over. But programmer expectations have risen as quickly, if not faster. They expect their editors to reparse files in milliseconds after every keystroke.

** 要快。** 计算机的速度比最初发明解析器技术时快了几千倍。那种需要优化解析器，以便它能在喝咖啡的时候处理完整个源文件的日子已经一去不复返了。但是程序员的期望值也上升得同样快，甚至更快。他们希望他们的编辑器能在每次击键后的几毫秒内回复文件。

- > **Report as many distinct errors as there are.** Aborting after the first error is easy to implement, but it's annoying for users if every time they fix what they think is the one error in a file, a new one appears. They want to see them all.

** 尽可能多地报告出不同的错误 **。在第一个错误后中止是很容易实现的，但是如果每次当用户修复文件中的一个错误时，又出现了另一个新的错误，这对用户来说是很烦人的。他们希望一次看到所有的错误。

- > **Minimize cascaded errors.** Once a single error is found, the parser no longer really knows what's going on. It tries to get itself back on track and keep going, but if it gets confused, it may report a slew of ghost errors that don't indicate other real problems in the code. When the first error is fixed, those phantoms disappear, because they reflect only the parser's own confusion. Cascaded errors are annoying because they can scare the user into thinking their code is in a worse state than it is.

** 最小化 * 级联 * 错误 **。一旦发现一个错误，解析器就不再能知道发生了什么。它会试图让自己回到正轨并继续工作，但如果它感到混乱，它可能会报告大量的幽灵错误，而这些错误并不表明代码中存在其它问题。当第一个错误被修正后，这些幽灵错误

就消失了，因为它们只反映了解析器自身的混乱。级联错误很烦人，因为它们会让用户害怕，让用户认为自己的代码比实际情况更糟糕。

最后两点是相互矛盾的。我们希望尽可能多地报告单独的错误，但我们不想报告那些只是由早期错误的副作用导致的错误。

解析器对一个错误做出反应，并继续去寻找后面的错误的方式叫做 ** 错误恢复 **。这在 60 年代是一个热门的研究课题。那时，你需要把一叠穿孔卡片交给秘书，第二天再来看看编译器是否成功。在迭代循环如此缓慢的情况下，你真的会想在一次传递中找到代码中的每个错误。

如今，解析器在您甚至还没有完成输入之前就完成解析了，这不再是一个问题。简单，快速的错误恢复就可以了。

6.3.1 恐慌模式错误恢复

在过去设计的所有恢复技术中，最能经受住时间考验的一种叫做 ** 恐慌模式 **（有点令人震惊）。一旦解析器检测到一个错误，它就会进入恐慌模式。它知道至少有一个 token 是没有意义的，因为它目前的状态是在一些语法生成式的堆栈中间。

在程序继续进行解析之前，它需要将自己的状态和即将到来的标记序列对齐，使下一个标记能够匹配正则解析的规则。这个过程称为 ** 同步 **。

为此，我们在语法中选择一些规则来标记同步点。解析器会跳出所有嵌套的生成式直到回退至该规则中，来修复其解析状态。然后，它会丢弃令牌，直到遇到一个可以匹配该规则的标记，以此来同步标记流。

这些被丢弃的标记中隐藏的其它真正的语法错误都不会被报告，但是这也意味着由初始错误引起的其它级联错误也不会被 * 错误地 * 报告出来，这是个不错的权衡。

语法中传统的要同步的地方是语句之间。我们还没有这些，所以我们不会在这一章中真正地同步，但我们会在以后把这些机制准备好。

6.3.2 进入恐慌模式

在我们讨论错误恢复之前，我们正在编写解析括号表达式的代码。在解析表达式之后，会调用 ‘consume()‘方法查找收尾的 ‘)‘。这里，终于可以实现那个方法了：

<u>lox/Parser.java，在 match() 方法后添加：</u>

```
private Token consume(TokenType type, String message) {
 if (check(type)) return advance();


 throw error(peek(), message);
}
```

它和 ‘match() ‘方法类似，检查下一个标记是否是预期的类型。如果是，它就会消费该标记，一切都很顺利。如果是其它的标记，那么我们就遇到了错误。我们通过调用下面的方法来报告错误：

<u>lox/Parser.java，在 previous() 方法后添加：</u>

```
  private ParseError error(Token token, String message) {
   Lox.error(token, message);
   return new ParseError();
  }
```

首先，通过调用下面的方法向用户展示错误信息：

<u>lox/Lox.java，在 report() 方法后添加：</u>

```
  static void error(Token token, String message) {
   if (token.type == TokenType.EOF) {
    report(token.line, " at end", message);
   } else {
    report(token.line, " at '" + token.lexeme + "'", message);
   }
  }
```

该方法会报告给定标记处的错误。它显示了标记的位置和标记本身。这在以后会派上用场，因为我们在整个解释器中使用标记来跟踪代码中的位置。

在我们报告错误后，用户知道了他们的错误，但接下来解析器要做什么呢？回到 ‘error()‘方法中，我们创建并返回了一个 ‘ParseError‘，是下面这个新类的实例：

<u>lox/Parser.java，在 Parser 中嵌入内部类：</u>

```
class Parser {
 // 新增部分开始
^^Iprivate static class ParseError extends RuntimeException {}
 // 新增部分结束
 private final List<Token> tokens;
```

这是一个简单的哨兵类，我们用它来帮助解析器摆脱错误。‘error()‘方法是 * 返回 * 错误而不是 * 抛出 * 错误，因为我们希望解析器内的调用方法决定是否要跳脱出该错误。有些解析错误发生在解析器不可能进入异常状态的地方，这时我们就不需要同步。在这些地方，我们只需要报告错误，然后继续解析。

例如，Lox 限制了你可以传递给一个函数的参数数量。如果你传递的参数太多，解析器需要报告这个错误，但它可以而且应该继续解析额外的参数，而不是惊慌失措，进入恐慌模式。

但是，在我们的例子中，语法错误非常严重，以至于我们要进入恐慌模式并进行同步。丢弃标记非常简单，但是我们如何同步解析器自己的状态呢？

6.3.3 同步递归下降解析器

在递归下降中，解析器的状态（即它正在识别哪个规则）不是显式存储在字段中的。相反，我们使用 Java 自身的调用栈来跟踪解析器正在做什么。每一条正在被解析的规则都是栈上的一个调用帧。为了重置状态，我们需要清除这些调用帧。

在 Java 中，最自然的实现方式是异常。当我们想要同步时，我们抛出 ParseError 对象。在我们正同步的语法规则的方法上层，我们将捕获它。因为我们在语句边界上同步，所以我们可以在那里捕获异常。捕获异常后，解析器就处于正确的状态。剩下的就是同步标记了。

我们想要丢弃令牌，直至达到下一条语句的开头。这个边界很容易发现——这也是我们选其作为边界的原因。在 * 分号 * 之后，我们可能就结束了一条语句。大多数语句都通过一个关键字开头——‘for‘、‘if‘、‘return‘、‘var‘等等。当下一个标记是其中之一时，我们可能就要开始一条新语句了。

> This method encapsulates that logic:

下面的方法封装了这个逻辑：

<u>lox/Parser.java，在 error() 方法后添加：</u>

```
  private void synchronize() {
    advance();

    while (!isAtEnd()) {
      if (previous().type == SEMICOLON) return;

      switch (peek().type) {
        case CLASS:
        case FUN:
        case VAR:
        case FOR:
        case IF:
        case WHILE:
        case PRINT:
        case RETURN:
          return;
      }

      advance();
    }
  }
```

该方法会不断丢弃标记，直到它发现一个语句的边界。在捕获一个 ParseError 后，我

们会调用该方法，然后我们就有望回到同步状态。当它工作顺利时，我们就已经丢弃了无论如何都可能会引起级联错误的语法标记，现在我们可以从下一条语句开始解析文件的其余部分。

唉，我们还没有看到这个方法的实际应用，因为我们目前还没有语句。我们会在后面几章中开始引入语句。现在，如果出现错误，我们就会进入恐慌模式，一直跳出到最顶层，并停止解析。由于我们只能解析一个表达式，所以这并不是什么大损失。

6.4 调整解析器

我们现在基本上已经完成了对表达式的解析。我们还需要在另一个地方添加一些错误处理。当解析器在每个语法规则的解析方法中下降时，它最终会进入 ‘primary()‘。如果该方法中的 case 都不匹配，就意味着我们正面对一个不是表达式开头的语法标记。我们也需要处理这个错误。

<u>lox/Parser.java，在 primary() 方法中添加：</u>

```
    if (match(LEFT_PAREN)) {
      Expr expr = expression();
      consume(RIGHT_PAREN, "Expect ')' after expression.");
      return new Expr.Grouping(expr);
    }
^^I^^I// 新增部分开头
    throw error(peek(), "Expect expression.");
^^I^^I// 新增部分结束
  }
```

> With that, all that remains in the parser is to define an initial method to kick it off. That method is called, naturally enough, ‘parse()‘.

这样，解析器中剩下的工作就是定义一个初始方法来启动它。这个方法自然应该叫做 ‘parse()‘。

<u>lox/Parser.java，在 Parser() 方法后添加：</u>

```
  Expr parse() {
    try {
      return expression();
    } catch (ParseError error) {
      return null;
    }
  }
```

稍后在向语言中添加语句时，我们将重新审视这个方法。目前，它只解析一个表达式并返回它。我们还有一些临时代码用于退出恐慌模式。语法错误恢复是解析器的工作，

所以我们不希望 ParseError 异常逃逸到解释器的其它部分。

当确实出现语法错误时，该方法会返回 ‘null‘。这没关系。解析器承诺不会因为无效语法而崩溃或挂起，但它不承诺在发现错误时返回一个 * 可用的语法树 *。一旦解析器报告错误，就会对 ‘hadError‘赋值，然后跳过后续阶段。

最后，我们可以将全新的解析器挂到 Lox 主类并进行试验。我们仍然还没有解释器，所以现在，我们将表达式解析为一个语法树，然后使用上一章中的 AstPrinter 类来显示它。

删除打印已扫描标记的旧代码，将其替换为：

<u>lox/Lox.java，在 run() 方法中，替换其中 5 行 </u>

```
    List<Token> tokens = scanner.scanTokens();
    // 替换部分开始
    Parser parser = new Parser(tokens);
    Expr expression = parser.parse();

    // Stop if there was a syntax error.
    if (hadError) return;

    System.out.println(new AstPrinter().print(expression));
    // 替换部分结束
  }
```

祝贺你，你已经跨过了门槛！这就是手写解析器的全部内容。我们将在后面的章节中扩展赋值、语句和其它特性对应的语法，但这些都不会比我们本章处理的二元操作符更复杂。

启动解释器并输入一些表达式。查看它是如何正确处理优先级和结合性的？这对于不到 200 行代码来说已经很不错了。

6.5 挑战

> 1、In C, a block is a statement form that allows you to pack a series of statements where a single one is expected. The [comma operator](https://en.wikipedia.org/wiki/Comma_operator) is an analogous syntax for separated series of expressions can be given where a single expression is expected (except inside a function call's $Add support for comma expressions. Give them the same precedence and associativity as in C. Write the gramma$

1、在 C 语言中，块是一种语句形式，它允许你把一系列的语句打包作为一个语句来使用。[逗号运算符](https://en.wikipedia.org/wiki/Comma_operator)()

添加对逗号表达式的支持。赋予它们与 c 语言中相同的优先级和结合性。编写语法，然后实现必要的解析代码。

> 2、Likewise, add support for the C-style conditional or “ternary” operator ‘?:‘. What

precedence level is allowed between the ‘¿ and ‘:‘? Is the whole operator left-associative or right-associative?

2、同样，添加对 C 风格的条件操作符或" 三元" 操作符 ‘?:‘的支持。在 ‘?‘和 ‘:‘之间采用什么优先级顺序？整个操作符是左关联还是右关联？

> 3、Add error productions to handle each binary operator appearing without a left-hand operand. In other words, detect a binary operator appearing at the beginning of an expression. Report that as an error, but also parse and discard a right-hand operand with the appropriate precedence.

3、添加错误生成式处理没有左操作数的二元操作符。换句话说，检测出现在表达式开头的二元操作符。将其作为错误报告给用户，同时也要解析并丢弃具有相应优先级的右操作数。

——

> DESIGN NOTE: LOGIC VERSUS HISTORY

> Let’ s say we decide to add bitwise ‘‘ and ‘|‘ operators to Lox. Where should we put them in the precedence hierarchy? C—and most languages that follow in C’ s footsteps—place them below ‘==‘. This is widely considered a mistake because it means common operations like testing a flag require parentheses. > >

```
    List<Token> tokens = scanner.scanTokens();
    // 替换部分开始
    Parser parser = new Parser(tokens);
    Expr expression = parser.parse();

    // Stop if there was a syntax error.
    if (hadError) return;

    System.out.println(new AstPrinter().print(expression));
    // 替换部分结束
  }
```

6.6 设计笔记：逻辑和历史

假设我们决定在 Lox 中添加位元 ‘‘和 ‘|‘运算符。我们应该将它们放在优先级层次结构的哪个位置？C（以及大多数跟随 C 语言步伐的语言）将它们放在 ‘==‘之下。目前普遍认为这是一个错误，因为这意味着检测标志位等常用操作都需要加括号。

```
package com.craftinginterpreters.lox;
```