#2.领土地图 A Map of the Territory

> You must have a map, no matter how rough. Otherwise you wander all over the place. In “The Lord of the Rings” I never made anyone go farther than he could on a given day.
>
> ——J.R.R. Tolkien

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。

> We don’t want to wander all over the place, so before we set off, let’s scan the territory charted by previous language implementers. It will help us understand where we are going and alternate routes others take.

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。

> First, let me establish a shorthand. Much of this book is about a language’s *implementation*, which is distinct from the *language itself* in some sort of Platonic ideal form. Things like “stack”, “bytecode”, and “recursive descent”, are nuts and bolts one particular implementation might use. From the user’s perspective, as long as the resulting contraption faithfully follows the language’s specification, it’s all implementation detail.

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。

> We're going to spend a lot of time on those details, so if I have to write "language *implementation*" every single time I mention them, I'll wear my fingers off. Instead, I'll use "language" to refer to either a language or an implementation of it, or both, unless the distinction matters.

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 The Parts of a Language

#2.1 语言的各部分

> Engineers have been building programming languages since the Dark Ages of computing. As soon as we could talk to computers, we discovered doing so was too hard, and we enlisted their help. I find it fascinating that even though today's machines are literally a million times faster and have orders of magnitude more storage, the way we build programming languages is virtually unchanged.

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。

> Though the area explored by language designers is vast, the trails they've carved through it are few. Not every language takes the exact same path—some take a shortcut or two—but otherwise they are reassuringly similar from Rear Admiral Grace Hopper's first COBOL compiler all the way to some hot new transpile-to-JavaScript language whose "documentation"

> consists entirely of a single poorly-edited README in a Git repository somewhere.

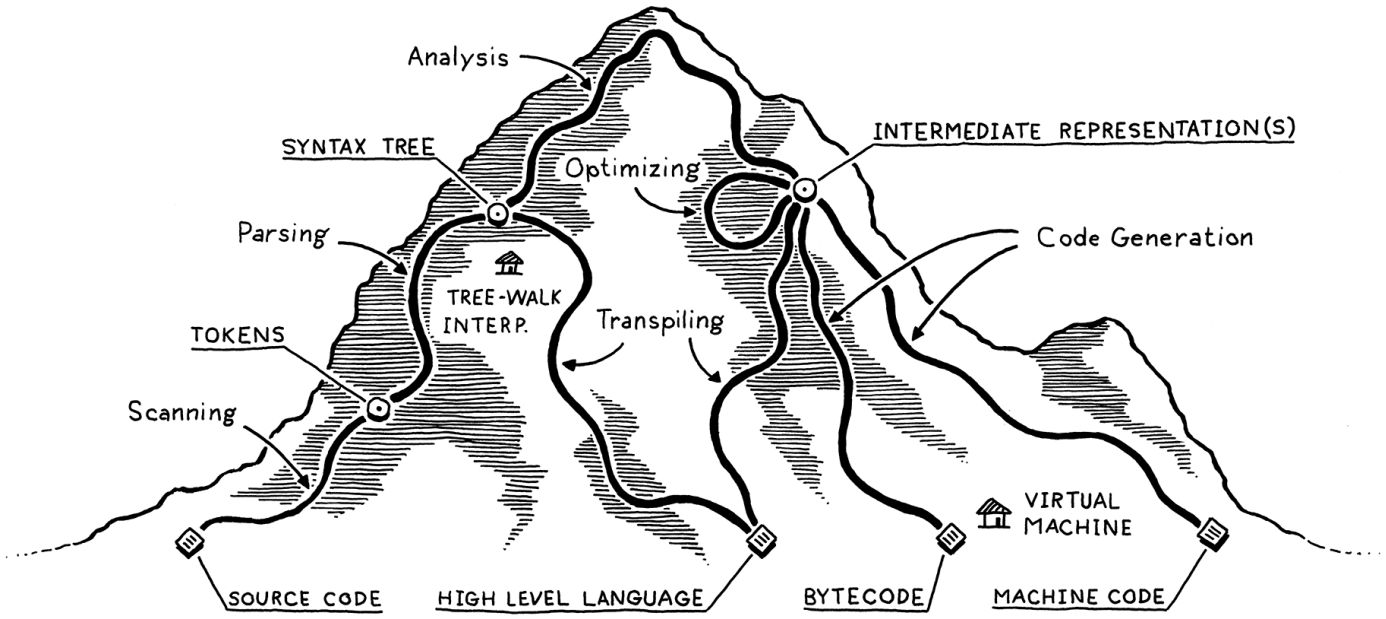
尽管语言设计师所探索的领域辽阔，但他们往往都走到相似的几条路上。 并非每种语言都采用完全相同的路径（有些会采用一种或两种捷径），但除此之外，从海军少将Grace Hopper的第一个COBOL编译器，一直到一些热门的新移植到JavaScript的语言（JS的 "文档 "甚至完全是由Git仓库中一个编辑得很差的README组成的[1]），都呈现出相似的特征，这令人十分欣慰。

> I visualize the network of paths an implementation may choose as climbing a mountain. You start off at the bottom with the program as raw source text, literally just a string of characters. Each phase analyzes the program and transforms it to some higher-level representation where the semantics—what the author wants the computer to do—becomes more apparent.

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。

> Eventually we reach the peak. We have a bird's-eye view of the users's program and can see what their code *means*. We begin our descent down the other side of the mountain. We transform this highest-level representation down to successively lower-level forms to get closer and closer to something we know how to make the CPU actually execute.

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。



> Let's trace through each of those trails and points of interest. Our journey begins on the left with the bare text of the user's source code:

让我们开始遍历所有有趣的路线和地点。我们的旅程从左边的用户源代码的纯文本开始

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

> ### #2.1.1 Scanning

#2.1.1 扫描

> The first step is **scanning**, also known as **lexing**, or (if you're trying to impress someone) **lexical analysis**. They all mean pretty much the same thing. I like "lexing" because it sounds like something an evil supervillain would do, but I'll use "scanning" because it seems to be marginally more commonplace.

第一步是**扫描**，也就是所谓的**词法分析**（lexing 或者强调写法 lexical analysis）。扫描和词法分析的意思相近。我喜欢词法分析，因为这听起来像是一个邪恶

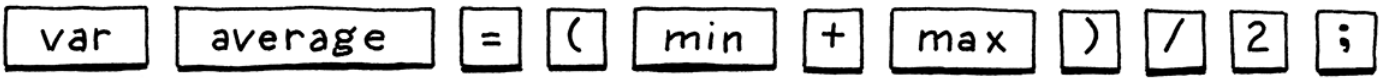
的超级大坏蛋会做的事情，但我还是用扫描，因为它似乎更常见一些。

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

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**词法单元**。有些词法单元是单个字符，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`）。

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

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词法单元序列。



> ### #2.1.2 Parsing

#2.1.2 语法分析

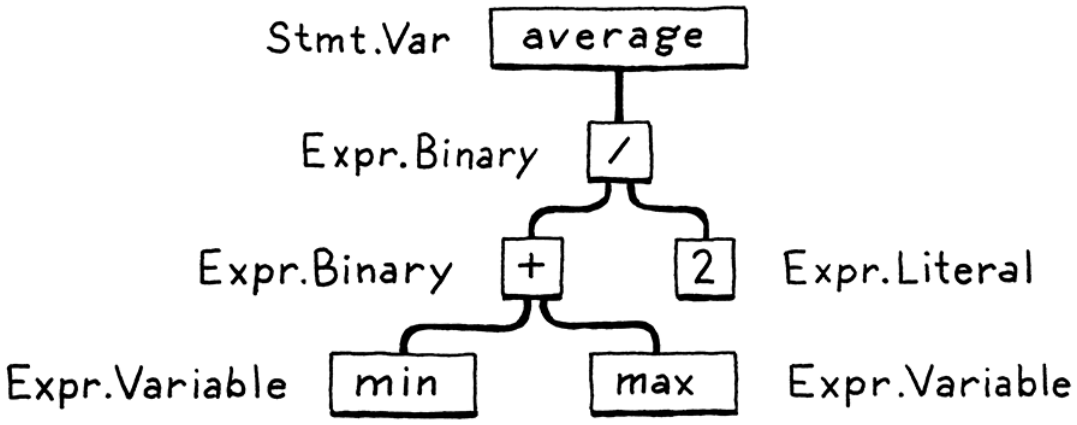
> The next step is **parsing**. This is where our syntax gets a **grammar**—the ability to compose larger expressions and statements out of smaller parts. Did you ever diagram sentences in English class? If so, you’ve done what a parser does, except that English has thousands and thousands of “key-

> words" and an overflowing cornucopia of ambiguity. Programming languages are much simpler.

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。你在英语课上做过语法图解吗？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。

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

解析器将扁平的词法单元序列转化为树形结构，树形结构能更好地反映语法的嵌套本质。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**"。



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

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被人工智能研究人员用于解析人类语言，人工智能研究人员试图通过这些技术让计算机能与我们对话。

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

事实证明，人类语言对于只能处理严密语法的解析器来说太混乱了，但面对编程语言这种简单的人造语法时，解析器表现得十分合适。唉，可惜我们这些有缺陷的人类在使用这些简单的语法时，仍然会不停地出错，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 Static analysis

#2.1.3 静态分析

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

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。

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

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？

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

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。

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

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**[2]。

> Take a deep breath. We have attained the summit of the mountain and a sweeping view of the user's program. All this semantic insight that is visible to us from analysis needs to be stored somewhere. There are a few places we can squirrel it away:
>
> - Often, it gets stored right back as **attributes** on the syntax tree itself—extra fields in the nodes that aren't initialized during parsing but get filled in later.
> - Other times, we may store data in a look-up table off to the side. Typically, the keys to this table are identifiers—names of variables and

> declarations. In that case, we call it a **symbol table** and the values it associates with each key tell us what that identifier refers to.
>
> - The most powerful bookkeeping tool is to transform the tree into an entirely new data structure that more directly expresses the semantics of the code. That's the next section.

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。从分析中可见的所有语义信息都需要存储在某个地方。我们可以把它存储在几个地方：

- 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。

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

到目前为止，所有内容都被视为实现的**前端**。 你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 Intermediate representations

#2.1.4 中介码

> You can think of the compiler as a pipeline where each stage's job is to organize the data representing the user's code in a way that makes the next stage simpler to implement. The front end of the pipeline is specific to the source language the program is written in. The back end is concerned with the final architecture where the program will run.

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。

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

在中间阶段，代码可能被存储在一些**中间代码**（**intermediate representation**， 也叫**IR**）中，这些中间代码与源文件或目的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[3]。

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

这可以让你更轻松地支持多种源语言和目标平台。假设你想在x86、ARM、SPARC 平台上实现Pascal、C和Fortran编译器。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[4]。

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

一个共享的中间代码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。

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

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 Optimization

#2.1.5 优化

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

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**。

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

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);
```

> We can do all of that arithmetic in the compiler and change the code to:

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;
```

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

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。有的时候这也会变成一种痴迷, 无法自拔。

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

我们在本书中通常会跳过这些棘手问题。令人惊讶的是许多成功的语言很少进行编译时优化。 例如，Lua和CPython生成相对未优化的代码，并将其大部分性能工作集中在运行时上[5]。

> ### #2.1.6 Code generation

#2.1.6 代码生成

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

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"。

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

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。

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

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，复杂的管线和足够塞满一架747行李舱的历史包袱。

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

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。追朔到上世纪60年代计算机体系结构 “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[6]。

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

为了解决这个问题，专家开始让他们的编译器生成虚拟机代码，包括BCPL的设计者Martin Richards以及Pascal设计者Niklaus Wirth。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。

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

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 Virtual machine

#2.1.7 虚拟机

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

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，你有两个选择。你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[7]。你仍然需要针对你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流水线的其余部分。你基本上是把你的字节码作为一种中间代码。

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

或者，你可以编写**虚拟机（VM）**[8]，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 Runtime

#2.1.8 运行时

> We have finally hammered the user's program into a form that we can execute. The last step is running it. If we compiled it to machine code, we simply tell the operating system to load the executable and off it goes. If we compiled it to bytecode, we need to start up the VM and load the program into that.

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。

> In both cases, for all but the basest of low-level languages, we usually need some services that our language provides while the program is running. For example, if the language automatically manages memory, we need a garbage collector going in order to reclaim unused bits. If our language supports "instance of" tests so you can see what kind of object you have, then we need some representation to keep track of the type of each object during execution.

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用 "instance of "测试我们拥有什么类型的对象，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。

> All of this stuff is going at runtime, so it's called, appropriately, the **runtime**. In a fully compiled language, the code implementing the runtime gets inserted directly into the resulting executable. In, say, Go, each compiled application has its own copy of Go's runtime directly embedded in it. If the language is run inside an interpreter or VM, then the runtime lives there. This is how most implementations of languages like Java, Python, and JavaScript work.

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 Shortcuts and Alternate Routes

#2.2 捷径和备选路线

> That's the long path covering every possible phase you might implement. Many languages do walk the entire route, but there are a few shortcuts and alternate paths.

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 Single-pass compilers

#2.2.1 单遍编译器

> Some simple compilers interleave parsing, analysis, and code generation so that they produce output code directly in the parser, without ever allocating any syntax trees or other IRs. These **single-pass compilers** restrict the design of the language. You have no intermediate data structures to store global information about the program, and you don't revisit any previously parsed part of the code. That means as soon as you see some expression, you need to know enough to correctly compile it.

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。你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，一旦你看到某个表达式，就需要足够的知识来正确地对其进行编译[9]。

> Pascal and C were designed around this limitation. At the time, memory was so precious that a compiler might not even be able to hold an entire *source file* in memory, much less the whole program. This is why Pascal's grammar requires type declarations to appear first in a block. It's why in C you can't call a function above the code that defines it unless you have an explicit forward declaration that tells the compiler what it needs to know to generate code for a call to the later function.

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 Tree-walk interpreters

#2.2.2 树遍历解释器

> Some programming languages begin executing code right after parsing it to an AST (with maybe a bit of static analysis applied). To run the program, the interpreter traverses the syntax tree one branch and leaf at a time, evaluating each node as it goes.

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。

> This implementation style is common for student projects and little languages, but is not widely used for general-purpose languages since it tends to be slow. Some people use “interpreter” to mean only these kinds of implementations, but others define that word more generally, so I’ll use the inarguably explicit **“tree-walk interpreter”** to refer to these. Our first interpreter rolls this way.

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[10]。

> ### #2.2.3 Transpilers

#2.2.3 转译器

> Writing a complete back end for a language can be a lot of work. If you have some existing generic IR to target, you could bolt your front end onto that. Otherwise, it seems like you’re stuck. But what if you treated some other *source language* as if it were an intermediate representation?

为一种语言编写一个完整的后端可能需要大量的工作。 如果你有一些现有的通用IR作为目标，则可以将前端转换到该IR上。 否则，你可能会陷入困境。 但

是，如果你将某些其他*源语言*视为中间代码，该怎么办？

> You write a front end for your language. Then, in the back end, instead of doing all the work to *lower* the semantics to some primitive target language, you produce a string of valid source code for some other language that's about as high level as yours. Then, you use the existing compilation tools for *that* language as your escape route off the mountain and down to something you can execute.

你需要为你的语言编写一个前端。然后，在后端，你可以生成一份与你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，你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。

> They used to call this a **source-to-source compiler** or a **transcompiler**. After the rise of languages that compile to JavaScript in order to run in the browser, they've affected the hipster sobriquet **transpiler**.

人们过去称之为**源到源编译器**或**转换编译器**[11]。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**。

> While the first transcompiler translated one assembly language to another, today, most transpilers work on higher-level languages. After the viral spread of UNIX to machines various and sundry, there began a long tradition of compilers that produced C as their output language. C compilers were available everywhere UNIX was and produced efficient code, so targeting C was a good way to get your language running on a lot of architectures.

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广泛运行在各种各样的机器上之后，编译器开始长期以C作为输出语言。C编译器在UNIX存在的地方都可以使用，并能生成有效的代码，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 browsers are the “machines” of today, and their “machine code” is JavaScript, so these days it seems almost every language out there has a compiler that targets JS since that’s the main way to get your code running in a browser.

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[12]。

> The front end—scanner and parser—of a transpiler looks like other compilers. Then, if the source language is only a simple syntactic skin over the target language, it may skip analysis entirely and go straight to outputting the analogous syntax in the destination language.

转译器的前端（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在语法方面的换皮版本，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。

> If the two languages are more semantically different, then you’ll see more of the typical phases of a full compiler including analysis and possibly even optimization. Then, when it comes to code generation, instead of outputting some binary language like machine code, you produce a string of grammatically correct source (well, destination) code in the target language.

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，而是生成一串语法正确的目标语言的源码（好吧，目标代码）。

> Either way, you then run that resulting code through the output language’s existing compilation pipeline and you’re good to go.

不管是哪种方式，你再通过目标语言已有的编译流水线运行生成的代码就可以了。

> #2.2.4 Just-in-time compilation

#2.2.4 即时编译

> This last one is less of a shortcut and more a dangerous alpine scramble best reserved for experts. The fastest way to execute code is by compiling it to machine code, but you might not know what architecture your end user's machine supports. What to do?

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?

> You can do the same thing that the HotSpot JVM, Microsoft's CLR and most JavaScript interpreters do. On the end user's machine, when the program is loaded—either from source in the case of JS, or platform-independent bytecode for the JVM and CLR—you compile it to native for the architecture their computer supports. Naturally enough, this is called **just-in-time compilation.** Most hackers just say "JIT", pronounced like it rhymes with "fit".

你可以做和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JS源代码还者是平台无关的JVM和CLR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。

> The most sophisticated JITs insert profiling hooks into the generated code to see which regions are most performance critical and what kind of data is flowing through them. Then, over time, they will automatically recompile those hot spots with more advanced optimizations.

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[13]。

> ## 2.3 Compilers and Interpreters

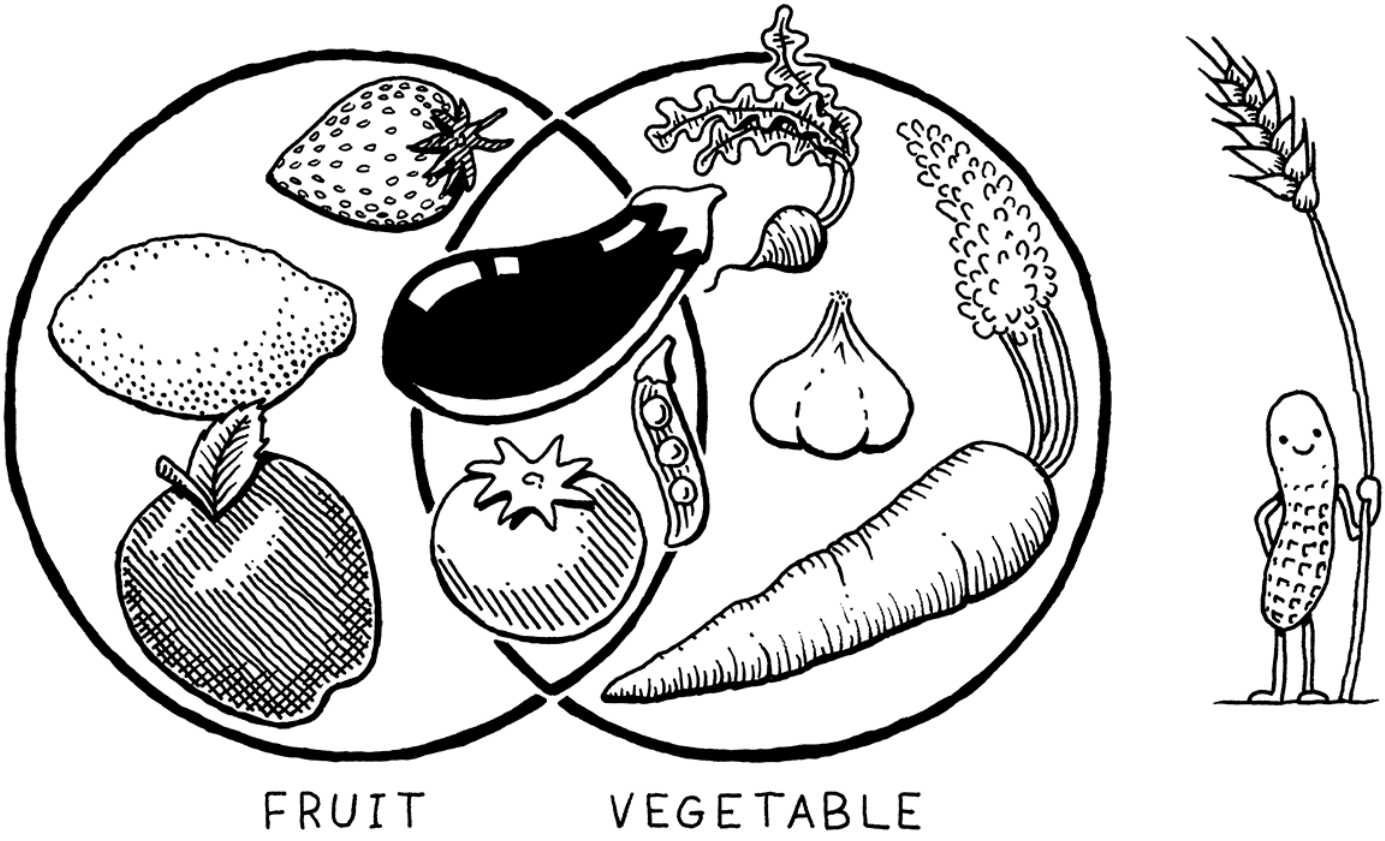
2.3 编译器和解释器

> Now that I've stuffed your head with a dictionary's worth of programming language jargon, we can finally address a question that's plagued coders since time immemorial: "What's the difference between a compiler and an interpreter?"

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?

> It turns out this is like asking the difference between a fruit and a vegetable. That seems like a binary either-or choice, but actually "fruit" is a *botanical* term and "vegetable" is *culinary*. One does not strictly imply the negation of the other. There are fruits that aren't vegetables (apples) and vegetables that are not fruits (carrots), but also edible plants that are both fruits *and* vegetables, like tomatoes.

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[14]。



> So, back to languages:
>
> - **Compiling** is an *implementation technique* that involves translating a source language to some other—usually lower-level—form. When you generate bytecode or machine code, you are compiling. When you transpile to another high-level language you are compiling too.
> - When we say a language implementation “is a **compiler**”, we mean it translates source code to some other form but doesn’t execute it. The user has to take the resulting output and run it themselves.
> - Conversely, when we say an implementation “is an **interpreter**”, we mean it takes in source code and executes it immediately. It runs programs “from source”.

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。

> Like apples and oranges, some implementations are clearly compilers and *not* interpreters. GCC and Clang take your C code and compile it to machine code. An end user runs that executable directly and may never even know which tool was used to compile it. So those are *compilers* for C.

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接受你的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*。

> In older versions of Matz' canonical implementation of Ruby, the user ran Ruby from source. The implementation parsed it and executed it directly by traversing the syntax tree. No other translation occurred, either internally or in any user-visible form. So this was definitely an *interpreter* for Ruby.

由 Matz 实现的老版本 Ruby 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*。

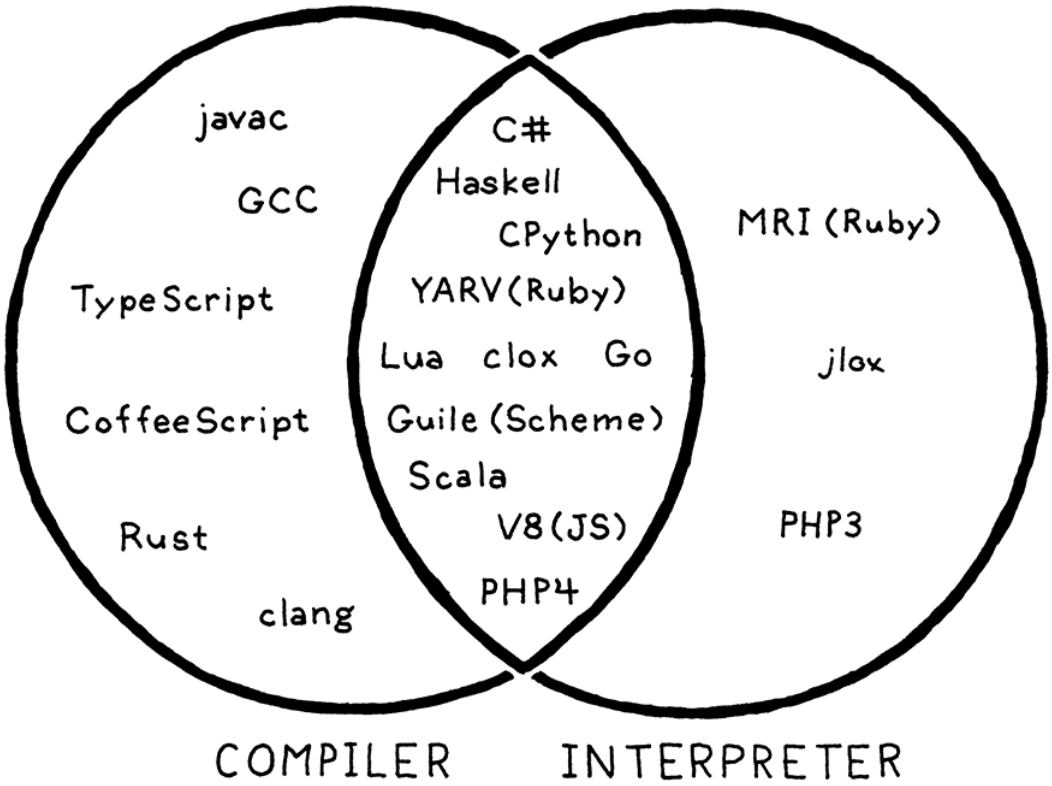
> But what of CPython? When you run your Python program using it, the code is parsed and converted to an internal bytecode format, which is then executed inside the VM. From the user's perspective, this is clearly an interpreter—they run their program from source. But if you look under CPython's scaly skin, you'll see that there is definitely some compiling going on.

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。

> The answer is that it is both. CPython *is* an interpreter, and it *has* a compiler. In practice, most scripting languages work this way, as you can see:

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[15]，如你所见：



> That overlapping region in the center is where our second interpreter lives too, since it internally compiles to bytecode. So while this book is nominally about interpreters, we'll cover some compilation too.

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 Our Journey

2.4 我们的旅程

> That's a lot to take in all at once. Don't worry. This isn't the chapter where you're expected to *understand* all of these pieces and parts. I just want you to know that they are out there and roughly how they fit together.

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。

> This map should serve you well as you explore the territory beyond the guided path we take in this book. I want to leave you yearning to strike out on your own and wander all over that mountain.

当你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。

> But, for now, it's time for our own journey to begin. Tighten your bootlaces, cinch up your pack, and come along. From here on out, all you need to focus on is the path in front of you.

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。

> ## CHALLENGES

习题

> 1、Pick an open source implementation of a language you like. Download the source code and poke around in it. Try to find the code that implements the scanner and parser. Are they hand-written, or generated using tools like Lex and Yacc? (`.l` or `.y` files usually imply the latter.)

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、Just-in-time compilation tends to be the fastest way to implement a dynamically-typed language, but not all of them use it. What reasons are there to *not* JIT?

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、Most Lisp implementations that compile to C also contain an interpreter that lets them execute Lisp code on the fly as well. Why?

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？

---

[^1]
毫无疑问，CS论文也有死胡同，被引为零的悲惨小众论文以及如今被遗忘的优化方法，这些优化方法只有在以单个字节为单位来衡量内存时才有意义。

[^2]
我们在本书中构建的语言是动态类型的，因此将在稍后的运行时中进行类型检查。

[^3]

有几种成熟的IR风格。点击你熟悉的搜索引擎，搜索 "控制流图"、"静态单赋值形式"、"延续传递形式 "和 "三位址码"。

[^4]

如果你曾经好奇GCC如何支持这么多疯狂的语言和体系结构，例如Motorola 68k上的Modula-3，现在你就明白了。 语言前端针对的是少数IR，主要是GIMPLE和RTL。 目标后端如68k，会接受这些IR并生成本机代码。

[^5]

如果你无法抗拒要进入这个领域，可以从以下关键字开始，例如“常量折叠”，“公共表达式消除”，“循环不变代码外提”，“全局值编号”，“强度降低”，“ 聚合量标量替换”，“死码删除”和“循环展开”。

[^6]

例如，AAD（"ASCII Adjust AX Before Division",除法前ASCII调整AX）指令可以让你执行除法，这听起来很有用。除了该指令将两个二进制编码的十进制数字作为操作数打包到一个16位寄存器中。你最后一次在16位机器上使用BCD是什么时候?

[^7]

这里的基本原则是，你把特定于体系架构的工作推得越靠后，你就可以在不同架构之间共享更多的早期阶段。不过，这里存在一些矛盾。 许多优化（例如寄存器分配和指令选择）在了解特定芯片的优势和功能时才能发挥最佳效果。 弄清楚编译器的哪些部分可以共享，哪些应该针对特定目标是一门艺术。

[^8]

术语“虚拟机”也指另一种抽象。 “系统虚拟机”在软件中模拟整个硬件平台和操作系统。 这就是你可以在Linux机器上玩Windows游戏的原因，也是云提供商为什么可以给客户提供控制自己的“服务器”的用户体验，而无需为每个用户实际分配单独的计算机。在本书中，我们将要讨论的虚拟机类型是“语言虚拟机”或“进程虚拟机”（如果你需要明确的话）。

[^9]

语法导向翻译是一种结构化的技术，用于构建这些一次性编译器。你可以将一个操作与语法的每个片段(通常是生成输出代码的语法片段)相关联。然

后，每当解析器匹配该语法块时，它就执行操作，一次构建一个规则的目标代码。

[^10]

一个明显的例外是早期版本的Ruby，它们是树遍历型解释器。在1.9时，Ruby的规范实现从最初的MRI（"Matz' Ruby Interpreter"）切换到了Koichi Sasada的YARV（"Yet Another Ruby VM"）。YARV是一个字节码虚拟机。

[^11]

第一个转编译器XLT86将8080程序集转换为8086程序集。 这看似简单，但请记住8080是8位芯片，而8086是16位芯片，可以将每个寄存器用作一对8位寄存器。 XLT86进行了数据流分析，以跟踪源程序中的寄存器使用情况，然后将其有效地映射到8086的寄存器集。它是由悲惨的计算机科学英雄加里·基尔达尔（Gary Kildall）撰写的。 他是最早认识到微型计算机前景的人之一，他创建了PL / M和CP / M，这是它们的第一种高级语言和操作系统。

[^12]

JS曾经是在浏览器中执行代码的唯一方式。多亏了Web Assembly，编译器现在有了第二种可以在Web上运行的低级语言。

[^13]

当然，这正是HotSpot JVM名称的来源。

[^14]

花生（连真正的坚果都算不上）和小麦等谷类其实都是水果，但我把这个图画错了。我能说什么呢，我是个软件工程师，不是植物学家。我也许应该抹掉这个花生小家伙，但他太可爱了，我不忍心。

[^15]

Go工具更是一个奇葩。如果你运行 `go build`，它就会把你的go源代码编译成机器代码然后停止。如果你输入 `go run`，它也会这样做，然后立即执行生成的可执行文件。所以，可以说go是一个编译器（你可以把它当做一个工具来编译代码而不运行）；也可以说是一个解释器（你可以调用它立即从源码中运行一个程序），并且有一个编译器（当你把它当做解释器使用时，它仍然在内部编译）。