#4.扫描 Scanning

> Take big bites. Anything worth doing is worth overdoing.
>
> —— Robert A. Heinlein, *Time Enough for Love*

大干特工。每件值得做的事都要尽力做好。

> The first step in any compiler or interpreter is scanning. The scanner takes in raw source code as a series of characters and groups it into a series of chunks we call **tokens**. These are the meaningful “words” and “punctuation” that make up the language’s grammar.

任何编译器或解释器的第一步都是扫描[1]。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。

> Scanning is a good starting point for us too because the code isn’t very hard—pretty much a `switch` statement with delusions of grandeur. It will help us warm up before we tackle some of the more interesting material later. By the end of this chapter, we’ll have a full-featured, fast scanner that can take any string of Lox source code and produce the tokens that we’ll feed into the parser in the next chapter.

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 The Interpreter Framework

#4.1 解释器框架

> Since this is our first real chapter, before we get to actually scanning some code we need to sketch out the basic shape of our interpreter, jlox. Everything starts with a class in Java.

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。

【译者注：原作者在代码的侧边栏标注了代码名及对应的操作（创建文件、追加代码、删除代码等），由于翻译版的格式受限，将这部分信息迁移到代码块之前，以带下划线的斜体突出，后同】

lox/Lox.java，创建新文件[2]

```
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();
    }
  }
}
```

> Stick that in a text file, and go get your IDE or Makefile or whatever set up. I'll be right here when you're ready. Good? OK!

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 is a scripting language, which means it executes directly from source. Our interpreter supports two ways of running code. If you start jlox from the command line and give it a path to a file, it reads the file and executes it.

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()));
}
```

> If you want a more intimate conversation with your interpreter, you can also run it interactively. Fire up jlox without any arguments, and it drops you into a prompt where you can enter and execute code one line at a time.

如果你想与你的解释器对话，可以交互式的启动它。 启动的时候不加任何参数就可以了，它会有一个提示符，你可以在提示符处一次输入并执行一行代码。

lox/Lox.java，添加到 `runFile()` 方法之后[3]

```
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);
    }
  }
```

> The `readLine()` function, as the name so helpfully implies, reads a line of input from the user on the command line and returns the result. To kill an interactive command-line app, you usually type Control-D. Doing so signals an “end-of-file” condition to the program. When that happens `readLine()` returns `null`, so we check for that to exit the loop.

`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。

> Both the prompt and the file runner are thin wrappers around this core function:

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);
      }
    }
```

> It's not super useful yet since we haven't written the interpreter, but baby steps, you know? Right now, it prints out the tokens our forthcoming scanner will emit so that we can see if we're making progress.

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 Error handling

#4.1.1 错误处理

> While we're setting things up, another key piece of infrastructure is *error handling*. Textbooks sometimes gloss over this because it's more a practical matter than a formal computer science-y problem. But if you care about making a language that's actually *usable*, then handling errors gracefully is vital.

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。

> The tools our language provides for dealing with errors make up a large portion of its user interface. When the user's code is working, they aren't thinking about our language at all—their headspace is all about *their program*. It's usually only when things go wrong that they notice our implementation.

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。

> When that happens, it's up to us to give the user all the information they need to understand what went wrong and guide them gently back to where they are trying to go. Doing that well means thinking about error handling all through the implementation of our interpreter, starting now.

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[4]。

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;
    }
```

> This `error()` function and its `report()` helper tells the user some syntax error occurred on a given line. That is really the bare minimum to be able to claim you even *have* error reporting. Imagine if you accidentally left a dangling comma in some function call and the interpreter printed out:

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!
```

> That's not very helpful. We need to at least point them to the right line. Even better would be the beginning and end column so they know *where* in the line. Even better than *that* is to *show* the user the offending line, like:

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.
```

> I'd love to implement something like that in this book but the honest truth is that it's a lot of grungy string manipulation code. Very useful for users, but not super fun to read in a book and not very technically interesting. So we'll stick with just a line number. In your own interpreters, please do as I say and not as I do.

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。

> The primary reason we're sticking this error reporting function in the main Lox class is because of that `hadError` field. It's defined here:

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;
```

> We'll use this to ensure we don't try to execute code that has a known error. Also, it lets us exit with a non-zero exit code like a good command line citizen should.

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);
    }
```

> We need to reset this flag in the interactive loop. If the user makes a mistake, it shouldn't kill their entire session.

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;
      }
```

> The other reason I pulled the error reporting out here instead of stuffing it into the scanner and other phases where the error might occur is to remind you that it's good engineering practice to separate the code that *generates* the errors from the code that *reports* them.

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。

> Various phases of the front end will detect errors, but it's not really their job to know how to present that to a user. In a full-featured language implementation, you will likely have multiple ways errors get displayed: on stderr, in

> an IDE's error window, logged to a file, etc. You don't want that code smeared all over your scanner and parser.

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。在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。

> Ideally, we would have an actual abstraction, some kind of "ErrorReporter" interface that gets passed to the scanner and parser so that we can swap out different reporting strategies. For our simple interpreter here, I didn't do that, but I did at least move the code for error reporting into a different class.

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[5]，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。

> With some rudimentary error handling in place, our application shell is ready. Once we have a Scanner class with a `scanTokens()` method, we can start running it. Before we get to that, let's get more precise about what tokens are.

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 Lexemes and Tokens

#4.2 词素和标记（词法单元）

下面是一行lox代码：

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

> Here, `var` is the keyword for declaring a variable. That three-character sequence “v-a-r” means something. But if we yank three letters out of the middle of `language`, like “g-u-a”, those don’t mean anything on their own.

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。

> That’s what lexical analysis is about. Our job is to scan through the list of characters and group them together into the smallest sequences that still represent something. Each of these blobs of characters is called a **lexeme**. In that example line of code, the lexemes are:

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：

> The lexemes are only the raw substrings of the source code. However, in the process of grouping character sequences into lexemes, we also stumble upon some other useful information. When we take the lexeme and bundle it together with that other data, the result is a token. It includes useful stuff like:

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 Token type

#4.2.1 标记类型

> Keywords are part of the shape of the language's grammar, so the parser often has code like, "If the next token is `while` then do . . . " That means the parser wants to know not just that it has a lexeme for some identifier, but that it has a *reserved* word, and *which* keyword it is.

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。

> The parser could categorize tokens from the raw lexeme by comparing the strings, but that's slow and kind of ugly. Instead, at the point that we recognize a lexeme, we also remember which *kind* of lexeme it represents. We have a different type for each keyword, operator, bit of punctuation, and literal type.

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[6]。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 Literal value

#4.2.2 字面量

> There are lexemes for literal values—numbers and strings and the like. Since the scanner has to walk each character in the literal to correctly identify it, it can also convert that textual representation of a value to the living runtime object that will be used by the interpreter later.

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 Location information

#4.2.3 位置信息

> Back when I was preaching the gospel about error handling, we saw that we need to tell users *where* errors occurred. Tracking that starts here. In

> our simple interpreter, we note only which line the token appears on, but more sophisticated implementations include the column and length too.

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[7]。

> We take all of this data and wrap it in a class.

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;
  }
}
```

> Now we have an object with enough structure to be useful for all of the later phases of the interpreter.

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 Regular Languages and Expressions

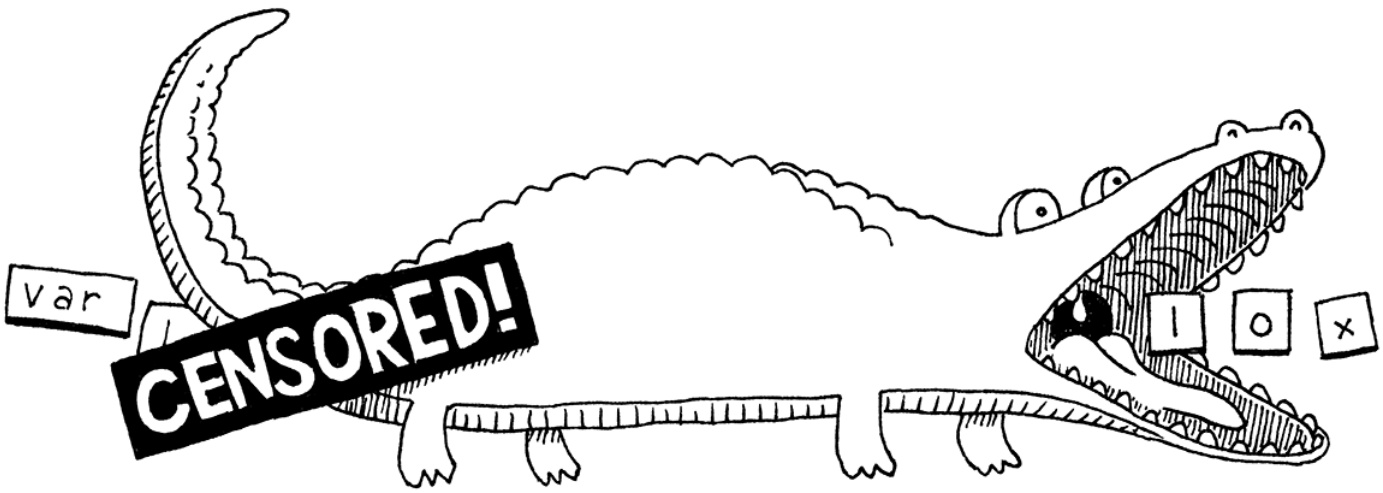
#4.3 正则语言和表达式

> Now that we know what we're trying to produce, let's, well, produce it. The core of the scanner is a loop. Starting at the first character of the source code, it figures out what lexeme it belongs to, and consumes it and any following characters that are part of that lexeme. When it reaches the end of that lexeme, it emits a token.

既然我们已知道我们要输出什么，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，扫描器会输出一个标记（词法单元 token）。

> Then it loops back and does it again, starting from the very next character in the source code. It keeps doing that, eating characters and occasionally, uh, excreting tokens, until it reaches the end of the input.

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，排出标记，直到它到达输入的终点。



> The part of the loop where we look at a handful of characters to figure out which kind of lexeme it “matches” may sound familiar. If you know regular expressions, you might consider defining a regex for each kind of lexeme and using those to match characters. For example, Lox has the same rules as C for identifiers (variable names and the like). This regex matches one:

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。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]*
```

> If you did think of regular expressions, your intuition is a deep one. The rules that determine how a particular language groups characters into lexemes are called its **lexical grammar**. In Lox, as in most programming languages, the rules of that grammar are simple enough for the language to be classified a **regular language**. That’s the same “regular” as in regular expressions.

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**[8]。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。

> You very precisely *can* recognize all of the different lexemes for Lox using regexes if you want to, and there’s a pile of interesting theory underlying why that is and what it means. Tools like Lex or Flex are designed expressly to let you do this—throw a handful of regexes at them, and they give you a complete scanner back.

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[9]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。

> Since our goal is to understand how a scanner does what it does, we won't be delegating that task. We're about handcrafted goods.

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 The Scanner Class

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[10]

```
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;
  }
}
```

> We store the raw source code as a simple string, and we have a list ready to fill with tokens we're going to generate. The aforementioned loop that does that looks like this:

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;
    }
```

> The scanner works its way through the source code, adding tokens until it runs out of characters. Then it appends one final "end of file" token. That isn't strictly needed, but it makes our parser a little cleaner.

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。

> This loop depends on a couple of fields to keep track of where the scanner is in the source code.

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。

lox/Scanner.java，在Scanner类中添加：

```
  private final List<Token> tokens = new ArrayList<>();
// 添加下面三行代码
  private int start = 0;
  private int current = 0;
  private int line = 1;

  Scanner(String source) {
```

> The `start` and `current` fields are offsets that index into the string. The `start` field points to the first character in the lexeme being scanned, and `current` points at the character currently being considered. The `line` field tracks what source line `current` is on so we can produce tokens that know their location.

`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。

> Then we have one little helper function that tells us if we've consumed all the characters.

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 Recognizing Lexemes

#4.5 识别词素

> In each turn of the loop, we scan a single token. This is the real heart of the scanner. We'll start simple. Imagine if every lexeme were only a single character long. All you would need to do is consume the next character and pick a token type for it. Several lexemes *are* only a single character in Lox, so let's start with those.

在每一次循环中，我们可以扫描出一个 token。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 token 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[11]。

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;
      }
    }
```

> Again, we need a couple of helper methods.

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));
  }
```

> The `advance()` method consumes the next character in the source file and returns it. Where `advance()` is for input, `addToken()` is for output. It grabs the text of the current lexeme and creates a new token for it. We'll use the other overload to handle tokens with literal values soon.

`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 token。我们马上会使用另一个重载方法来处理带有字面值的 token。

> ### #4.5.1 Lexical errors

#4.5.1 词法错误

> Before we get too far in, let's take a moment to think about errors at the lexical level. What happens if a user throws a source file containing some characters Lox doesn't use, like `@#^`, at our interpreter? Right now, those characters get silently discarded. They aren't used by the Lox language, but that doesn't mean the interpreter can pretend they aren't there. Instead, we report an error.

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;
      }
```

> Note that the erroneous character is still *consumed* by the earlier call to `advance()`. That's important so that we don't get stuck in an infinite loop.

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。

> Note also that we *keep scanning*. There may be other errors later in the program. It gives our users a better experience if we detect as many of those as possible in one go. Otherwise, they see one tiny error and fix it, only to have the next error appear, and so on. Syntax error Whac-A-Mole is no fun.

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。

> (Don't worry. Since `hadError` gets set, we'll never try to *execute* any of the code, even though we keep going and scan the rest of it.)

(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 Operators

#4.5.2 操作符

> We have single-character lexemes working, but that doesn't cover all of Lox's operators. What about `!`? It's a single character, right? Sometimes, yes, but if the very next character is an equals sign, then we should instead create a `!=` lexeme. Note that the `!` and `=` are *not* two independent operators. You can't write `! =` in Lox and have it behave like an inequality operator. That's why we need to scan `!=` as a single lexeme. Likewise, `<`, `>`, and `=` can all be followed by `=` to create the other equality and comparison operators.

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*两个独立的操作符。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。

> For all of these, we need to look at the second character.

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:
```

> Those cases use this new method:

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;
  }
```

> It's like a conditional `advance()`. We only consume the current character if it's what we're looking for.

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。

> Using `match()`, we recognize these lexemes in two stages. When we reach, for example, `!`, we jump to its switch case. That means we know the lexeme *starts* with `!`. Then we look at the next character to determine if we're on a `!=` or merely a `!`.

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 Longer Lexemes

#4.6 更长的词素

> We're still missing one operator: `/` for division. That character needs a little special handling because comments begin with a slash too.

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:
```

> This is similar to the other two-character operators, except that when we find a second `/`, we don't end the token yet. Instead, we keep consuming characters until we reach the end of the line.

这与其它的双字符操作符是类似的，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。

> This is our general strategy for handling longer lexemes. After we detect the beginning of one, we shunt over to some lexeme-specific code that keeps eating characters until it sees the end.

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。

> We've got another helper:

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);
  }
```

> It's sort of like `advance()`, but doesn't consume the character. This is called **lookahead**. Since it only looks at the current unconsumed character, we have *one character of lookahead*. The smaller this number is, generally, the faster the scanner runs. The rules of the lexical grammar dictate how much lookahead we need. Fortunately, most languages in wide use peek only one or two characters ahead.

这有点像 `advance()` 方法，只是不会消费字符。这就是所谓的**lookahead(前瞻)**[12]。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，前瞻的字符越少，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。

> Comments are lexemes, but they aren't meaningful, and the parser doesn't want to deal with them. So when we reach the end of the comment, we *don't* call `addToken()`. When we loop back around to start the next lexeme, `start` gets reset and the comment's lexeme disappears in a puff of smoke.

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。

> While we're at it, now's a good time to skip over those other meaningless characters: newlines and whitespace.

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.");
```

> When encountering whitespace, we simply go back to the beginning of the scan loop. That starts a new lexeme *after* the whitespace character. For newlines, we do the same thing, but we also increment the line counter. (This is why we used `peek()` to find the newline ending a comment instead of `match()`. We want that newline to get us here so we can update `line`.)

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)

> Our scanner is getting smarter. It can handle fairly free-form code like:

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 String literals

#4.6.1 字符串字面量

> Now that we're comfortable with longer lexemes, we're ready to tackle literals. We'll do strings first, since they always begin with a specific character, `"`.

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:
```

> That calls:

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);
    }
```

> Like with comments, we consume characters until we hit the `"` that ends the string. We also gracefully handle running out of input before the string is closed and report an error for that.

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。

> For no particular reason, Lox supports multi-line strings. There are pros and cons to that, but prohibiting them was a little more complex than allowing them, so I left them in. That does mean we also need to update `line` when we hit a newline inside a string.

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。

> Finally, the last interesting bit is that when we create the token, we also produce the actual string *value* that will be used later by the interpreter. Here, that conversion only requires a `substring()` to strip off the surrounding quotes. If Lox supported escape sequences like `\n`, we'd unescape those here.

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 Number literals

#4.6.2 数字字面量

> All numbers in Lox are floating point at runtime, but both integer and decimal literals are supported. A number literal is a series of digits optionally followed by a `.` and one or more trailing digits.

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[13]。

```
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.
```

> We could easily support the former, but I left it out to keep things simple. The latter gets weird if we ever want to allow methods on numbers like `123.sqrt()`.

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。

> To recognize the beginning of a number lexeme, we look for any digit. It's kind of tedious to add cases for every decimal digit, so we'll stuff it in the default case instead.

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()方法中替换一行：

```
      default:
        // 替换部分开始
```

```
        if (isDigit(c)) {
          number();
        } else {
          Lox.error(line, "Unexpected character.");
        }
        // 替换部分结束
        break;
```

> This relies on this little utility:

这里依赖下面的小工具函数[14]：

lox/Scanner.java，在 peek()方法之后添加：

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

> Once we know we are in a number, we branch to a separate method to consume the rest of the literal, like we do with strings.

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)));
    }
```

> We consume as many digits as we find for the integer part of the literal. Then we look for a fractional part, which is a decimal point (`.`) followed by at least one digit. If we do have a fractional part, again, we consume as many digits as we can find.

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。

> Looking past the decimal point requires a second character of lookahead since we don't want to consume the `.` until we're sure there is a digit *after* it. So we add:

在定位到小数点之后需要继续前瞻第二个字符，因为我们只有确认其*后*有数字才会消费 `.` 。所以我们添加了[15]:

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);
  }
```

> Finally, we convert the lexeme to its numeric value. Our interpreter uses Java's `Double` type to represent numbers, so we produce a value of that type. We're using Java's own parsing method to convert the lexeme to a real Java double. We could implement that ourselves, but, honestly, unless you're trying to cram for an upcoming programming interview, it's not worth your time.

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。

> The remaining literals are Booleans and `nil`, but we handle those as keywords, which gets us to . . .

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 Reserved Words and Identifiers

#4.7 保留字和标识符

> Our scanner is almost done. The only remaining pieces of the lexical grammar to implement are identifiers and their close cousins, the reserved words. You might think we could match keywords like `or` in the same way we handle multiple-character operators like `<=`.

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;
```

> Consider what would happen if a user named a variable `orchid`. The scanner would see the first two letters, `or`, and immediately emit an `or` keyword token. This gets us to an important principle called **maximal munch**. When two lexical grammar rules can both match a chunk of code that the scanner is looking at, *whichever one matches the most characters wins*.

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)[16]。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*。

> That rule states that if we can match `orchid` as an identifier and `or` as a keyword, then the former wins. This is also why we tacitly assumed, previously, that `<=` should be scanned as a single `<=` token and not `<` followed by `=`.

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 `=`。

> Maximal munch means we can't easily detect a reserved word until we've reached the end of what might instead be an identifier. After all, a reserved word *is* an identifier, it's just one that has been claimed by the language for its own use. That's where the term **reserved word** comes from.

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。

> So we begin by assuming any lexeme starting with a letter or underscore is an identifier.

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();
  // 新增部分开始
} else if (isAlpha(c)) {
```

```
        identifier();
      // 新增部分结束
      } else {
        Lox.error(line, "Unexpected character.");
      }
```

> The rest of the code lives over here:

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);
  }
```

> We define that in terms of these helpers:

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);
  }
```

> That gets identifiers working. To handle keywords, we see if the identifier's lexeme is one of the reserved words. If so, we use a token type specific to that keyword. We define the set of reserved words in a map.

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);
  }
```

> Then, after we scan an identifier, we check to see if it matches anything in the map.

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();

    // 替换部分开始
    String text = source.substring(start, current);
    TokenType type = keywords.get(text);
```

```
      if (type == null) type = IDENTIFIER;
      addToken(type);
      // 替换部分结束
    }
```

> If so, we use that keyword's token type. Otherwise, it's a regular user-defined identifier.

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。

> And with that, we now have a complete scanner for the entire Lox lexical grammar. Fire up the REPL and type in some valid and invalid code. Does it produce the tokens you expect? Try to come up with some interesting edge cases and see if it handles them as it should.

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。

> ## #CHALLENGES

#习题

> 1、The lexical grammars of Python and Haskell are not *regular*. What does that mean, and why aren't they?

1、Python和Haskell的语法不是*常规的*。 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、Aside from separating tokens—distinguishing `print foo` from `printfoo`—spaces aren't used for much in most languages. However, in a couple of dark corners, a space *does* affect how code is parsed in CoffeeScript, Ruby, and the C preprocessor. Where and what effect does it have in each of those languages?

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、Our scanner here, like most, discards comments and whitespace since those aren't needed by the parser. Why might you want to write a scanner that does *not* discard those? What would it be useful for?

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、Add support to Lox's scanner for C-style `/* ... */` block comments. Make sure to handle newlines in them. Consider allowing them to nest. Is adding support for nesting more work than you expected? Why?

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？

> ## #DESIGN NOTE: IMPLICIT SEMICOLONS

#设计笔记：隐藏的分号

> Programmers today are spoiled for choice in languages and have gotten picky about syntax. They want their language to look clean and modern. One bit of syntactic lichen that almost every new language scrapes off (and some ancient ones like BASIC never had) is `;` as an explicit statement terminator.
>
> Instead, they treat a newline as a statement terminator where it makes sense to do so. The “where it makes sense” part is the challenging bit. While *most* statements are on their own line, sometimes you need to spread a single statement across a couple of lines. Those intermingled newlines should not be treated as terminators.
>
> Most of the obvious cases where the newline should be ignored are easy to detect, but there are a handful of nasty ones:
>
> - A return value on the next line:
>
>   ```
>   if (condition) return
>   "value"
>   ```
>
>   Is “value” the value being returned, or do we have a `return` statement with no value followed by an expression statement containing a string literal?
>
> - A parenthesized expression on the next line:
>
>   ```
>   func
>   (parenthesized)
>   ```
>
>   Is this a call to `func(parenthesized)`, or two expression statements, one for `func` and one for a parenthesized expression?
>
> - A `-` on the next line:
>
>   ```
>   first
>   -second
>   ```

Is this `first - second` —an infix subtraction—or two expression statements, one for `first` and one to negate `second`?

In all of these, either treating the newline as a separator or not would both produce valid code, but possibly not the code the user wants. Across languages, there is an unsettling variety of rules used to decide which newlines are separators. Here are a couple:

- Lua completely ignores newlines, but carefully controls its grammar such that no separator between statements is needed at all in most cases. This is perfectly legit:

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

  Lua avoids the `return` problem by requiring a `return` statement to be the very last statement in a block. If there is a value after `return` before the keyword `end`, it *must* be for the `return`. For the other two cases, they allow an explicit `;` and expect users to use that. In practice, that almost never happens because there's no point in a parenthesized or unary negation expression statement.

- Go handles newlines in the scanner. If a newline appears following one of a handful of token types that are known to potentially end a statement, the newline is treated like a semicolon, otherwise it is ignored. The Go team provides a canonical code formatter, gofmt, and the ecosystem is fervent about its use, which ensures that idiomatic styled code works well with this simple rule.

- Python treats all newlines as significant unless an explicit backslash is used at the end of a line to continue it to the next line. However, newlines anywhere inside a pair of brackets (`()`, `[]`, or `{}`) are ignored. Idiomatic style strongly prefers the latter.

  This rule works well for Python because it is a highly statement-oriented language. In particular, Python's grammar ensures a statement never appears inside an expression. C does the same, but many other languages which have a "lambda" or function literal syntax do not.

> An example in JavaScript:
>
> ```
> console.log(function() {
>   statement();
> });
> ```
>
> Here, the `console.log()` *expression* contains a function literal which in turn contains the *statement* `statement();`.
>
> Python would need a different set of rules for implicitly joining lines if you could get back *into* a statement where newlines should become meaningful while still nested inside brackets.
>
> - JavaScript's "automatic semicolon insertion" rule is the real odd one. Where other languages assume most newlines *are* meaningful and only a few should be ignored in multi-line statements, JS assumes the opposite. It treats all of your newlines as meaningless whitespace *unless* it encounters a parse error. If it does, it goes back and tries turning the previous newline into a semicolon to get something grammatically valid.
>
>   This design note would turn into a design diatribe if I went into complete detail about how that even *works*, much less all the various ways that JavaScript's "solution" is a bad idea. It's a mess. JavaScript is the only language I know where many style guides demand explicit semicolons after every statement even though the language theoretically lets you elide them.
>
> If you're designing a new language, you almost surely *should* avoid an explicit statement terminator. Programmers are creatures of fashion like other humans, and semicolons are as passé as ALL CAPS KEYWORDS. Just make sure you pick a set of rules that make sense for your language's particular grammar and idioms. And don't do what JavaScript did.

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[lambda]。

- 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。JavaScript是我所知道的唯一（风格指南和语言本身背离）的语言，它的许多风格指南要求在每条语句后都显式地使用分号，但该语言却理论上允许您省略分号。

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。

---

[^1]

一直以来，这项工作被称为 "扫描(scanning) "和 "词法分析(lexing)"（ "词法分析(lexical analysis)"的简称）。早在计算机还像Winnebagos一样大，但内存比你的手表还小的时候，有些人就用 "扫描 "来指代从磁盘上读取原始源代码字符并在内存中缓冲的那段代码。然后，"lexing "是后续阶段，对字符做有用的操作。现在，将源文件读入内存是很平常的事情，因此在编译器中很少出现不同的阶段。 因此，这两个术语基本上可以互换。

[^2]

`System.exit(64)`，对于退出代码，我使用UNIX sysexts .h头文件中定义的约定。这是我能找到的最接近标准的东西。

[^3]

交互式提示符也被称为REPL(发音像rebel，但替换为p)。它的名称来自于Lisp，实现Lisp非常简单，只需围绕几个内置函数进行循环: `(print (eval (read)))` 从嵌套最内的调用向外执行，读取一行输入，求值，打印结果，然后循环并再次执行。

[^4]

说了这么多，对于这个解释器，我们要构建的只是基本框架。我很想谈谈交互式调试器、静态分析器和其它有趣的东西，但是篇幅实在有限。

[^5]

我第一次实现jlox的时候正是如此。最后我把它拆出去了，因为对于本书的最小解释器来说，这有点过度设计了。

[^6]

毕竟，字符串比较最终也会比对单个字符，这不正是扫描器的工作吗?

[^7]

一些标记实现将位置存储为两个数字：从源文件开始到词素开始的偏移量，以及词素的长度。扫描器无论如何都会知道这些数字，因此计算这些数字没有任何开销。通过回头查看源文件并计算前面的换行数，可以将偏移量转换为行和列位置。这听起来很慢，确实如此。然而，只有当你需要向用户实际显示行和列的时候，你才需要这样做。大多数标记从来不会出现在错误信息中。对于这些标记，你花在提前计算位置信息上的时间越少越好。

[^8]

我很痛心要对理论做这么多掩饰，尤其是当它像乔姆斯基谱系和有限状态机那样有趣的时候。但说实话，其他的书比我写得好。*Compilers: Principles, Techniques, and Tools*(常被称为“龙书”)是最经典的参考书。

[^9]

Lex是由Mike Lesk和Eric Schmidt创建的。是的，就是那个曾任谷歌执行董事长的Eric Schmidt。我并不是说编程语言是通往财富和名声的必经之路，但我们中至少已经有一位超级亿万富翁。

[^10]

我知道很多人认为静态导入是一种不好的代码风格，但这样我就不必在扫描器和解析器中到处写 `TokenType` 了。恕我直言，在一本书中，每个字符都很重要

[^11]

想知道这里为什么没有 `/` 吗？别担心，我们会解决的。

[^12]

技术上来说， `match()` 方法也是在做前瞻。 `advance()` 和 `peek()` 是基本运算符， `match()` 将它们结合起来。

[^13]

因为我们只会根据数字来判断数字字面量，这就意味着 `-123` 不是一个数字*字面量*。相反， `-123` 是一个*表达式*，将 `-` 应用到数字字面量 `123` 。在实践中，结果是一样的，尽管它有一个有趣的边缘情况。试想一下，如果我们要在数字上添加方法调用： `print -123.abs();` ，这里会输出 `-123` ，因为负号的优先级低于方法调用。我们可以通过将 `-` 作为数字字面值的一部分来解决这个问题。但接着考虑： `var n = 123; print -n.abs();` ，结果仍然是 `-123` ，所以现在语言似乎不一致。无论你怎么做，有些情况最后都会变得很奇怪。

[^14]

Java标准库中提供了Character.isDigit()，这似乎是个不错的选择。唉，该方法中还允许梵文数字、全宽数字和其他我们不想要的有趣的东西。

[^15]

我本可以让 `peek()` 方法接受一个参数来表示要前瞻的字符数，而不需要定义两个函数。但这样做就会允许前瞻任意长度的字符。提供两个函数可以让读者更清楚地知道，我们的扫描器最多只能向前看两个字符。

[^16]

看一下这段讨厌的C代码： `---a;` ，它有效吗？这取决于扫描器如何分割词素。如果扫描器看到的是 `- --a;` ，那它就可以被解析。但是这需要扫描器知道代码前后的语法结构，这比我们需要的更复杂。相反，最大匹配原则表明，扫描结果总是： `-- -a;` ，它就会这样扫描，尽管这样做会在解析器中导致后面的语法错误。

[^lambda]

现在你明白为什么Python中的 `lambda` 只允许单行的表达式体了吧。