#8.表达式和状态 Statements and State

> *All my life, my heart has yearned for a thing I cannot name.*
>
> —— André Breton, *Mad Love*

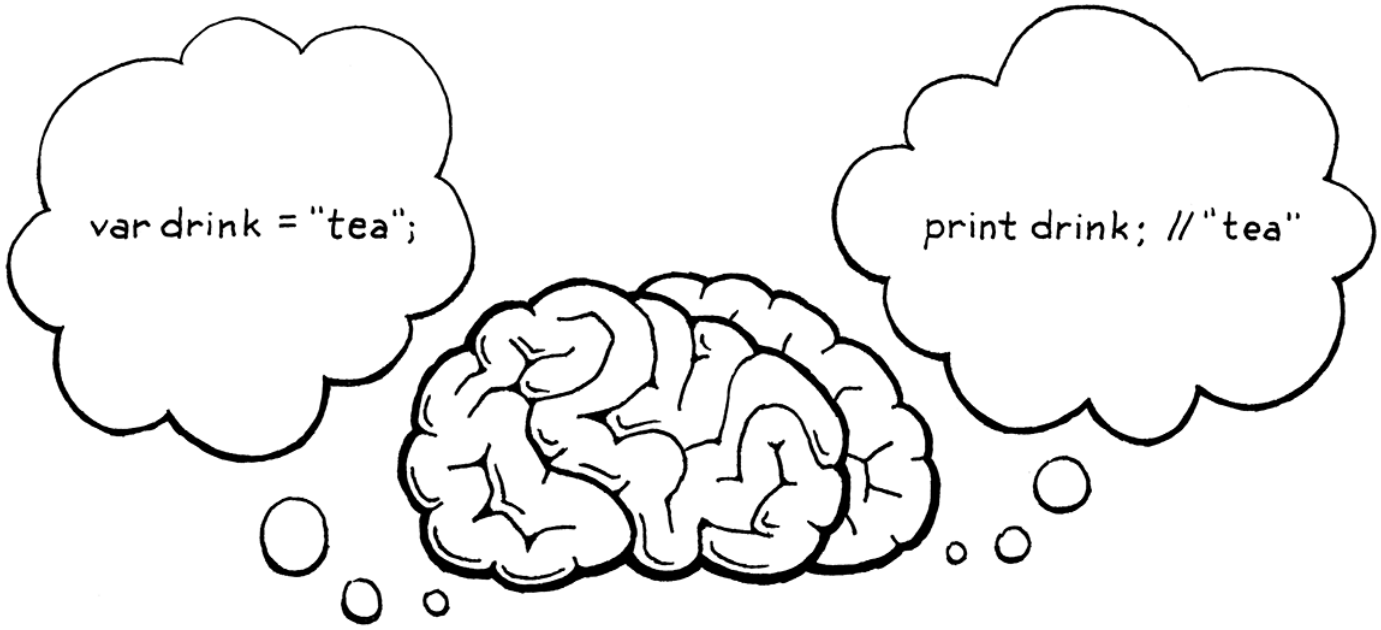
终我一生，我们的内心都在渴求一种我无法名状的东西。

> The interpreter we have so far feels less like programming a real language and more like punching buttons on a calculator. “Programming” to me means building up a system out of smaller pieces. We can’t do that yet because we have no way to bind a name to some data or function. We can’t compose software without a way to refer to the pieces.

到目前为止，我们提供解释器的感觉不太像是在使用一种真正的语言进行编程，更像是在计算器上按按钮。对我来说，"编程 "意味着用较小的部分构建出一个系统。我们目前还不支持这样做，因为我们还无法将一个名称绑定到某个数据或函数。我们不能在无法引用小片段的情况下编写软件。

> To support bindings, our interpreter needs internal state. When you define a variable at the beginning of the program and use it at the end, the interpreter has to hold on to the value of that variable in the meantime. So in this chapter, we will give our interpreter a brain that can not just process, but *remember*.

为了支持绑定，我们的解释器需要保存内部状态。如果你在程序开始处定义了一个变量，并在结束处使用它，那么解释器必须在这期间保持该变量的值。所以在这一章中，我们会给解释器一个大脑，它不仅可以运算，而且可以*记忆*。



> State and statements go hand in hand. Since statements, by definition, don’t evaluate to a value, they need to do something else to be useful. That something is called a **side effect**. It could mean producing user-visible output or modifying some state in the interpreter that can be detected later. The latter makes them a great fit for defining variables or other named entities.

状态和语句是相辅相成的。因为根据定义，语句不会计算出一个具体值，而是需要做一些事情来发挥作用。这些事情被称为**副作用(side effect)**。它可能意味着产生用户可见的输出，或者修改解释器中的一些状态，而这些状态后续可以被检测到。第二个特性使得语句非常适合于定义变量或其他命名实体。

> In this chapter, we’ll do all of that. We’ll define statements that produce output (`print`) and create state (`var`). We’ll add expressions to access and assign to variables. Finally, we’ll add blocks and local scope. That’s a lot to stuff into one chapter, but we’ll chew through it all one bite at a time.

在这一章中，我们会实现所有这些。我们会定义可以产生输出和创建状态的语句，然后会添加表达式来访问和赋值给这些变量，最后，我们会引入代码块和局部作用域。这一章要讲的内容太多了，但是我们会一点一点地把它们嚼碎。

> ## #8.1 Statements

#8.1 语句

> We start by extending Lox's grammar with statements. They aren't very different from expressions. We start with the two simplest kinds:

我们首先扩展Lox的语法以支持语句。 语句与表达式并没有很大的不同，我们从两种最简单的类型开始：

1. > An **expression statement** lets you place an expression where a statement is expected. They exist to evaluate expressions that have side effects. You may not notice them, but you use them all the time in C, Java, and other languages. Any time you see a function or method call followed by a `;`, you're looking at an expression statement.

   表达式语句可以让您将表达式放在需要语句的位置。它们的存在是为了计算有副作用的表达式。您可能没有注意到它们，但其实你在C、Java和其他语言中一直在使用表达式语句[1]。如果你看到一个函数或方法调用后面跟着一个`;`，您看到的其实就是一个表达式语句。

2. > A `print` **statement** evaluates an expression and displays the result to the user. I admit it's weird to bake printing right into the language instead of making it a library function. Doing so is a concession to the fact that we're building this interpreter one chapter at a time and want to be able to play with it before it's all done. To make print a library function, we'd have to wait until we had all of the machinery for defining and calling functions before we could witness any side effects.

   `print` **语句**会计算一个表达式，并将结果展示给用户。我承认把`print`直接放进语言中，而不是把它变成一个库函数，这很奇怪[2]。这样做是基于本书的

编排策略的让步，即我们会以章节为单位逐步构建这个解释器，并希望能够在完成解释器的所有功能之前能够使用它。如果让 `print` 成为一个标准库函数，我们必须等到拥有了定义和调用函数的所有机制之后，才能看到它发挥作用。

> New syntax means new grammar rules. In this chapter, we finally gain the ability to parse an entire Lox script. Since Lox is an imperative, dynamically typed language, the “top level” of a script is simply a list of statements. The new rules are:

新的词法意味着新的语法规则。在本章中，我们终于获得了解析整个Lox脚本的能力。由于Lox是一种命令式的、动态类型的语言，所以脚本的“顶层”也只是一组语句。新的规则如下：

```
program        → statement* EOF ;

statement      → exprStmt
               | printStmt ;

exprStmt       → expression ";" ;
printStmt      → "print" expression ";" ;
```

> The first rule is now `program`, which is the starting point for the grammar and represents a complete Lox script or REPL entry. A program is a list of statements followed by the special “end of file” token. The mandatory end token ensures the parser consumes the entire input and doesn’t silently ignore erroneous unconsumed tokens at the end of a script.

现在第一条规则是 `program`，这也是语法的起点，代表一个完整的Lox脚本或REPL输入项。程序是一个语句列表，后面跟着特殊的“文件结束”(EOF)标记。强制性的结束标记可以确保解析器能够消费所有输入内容，而不会默默地忽略脚本结尾处错误的、未消耗的标记。

> Right now, `statement` only has two cases for the two kinds of statements we’ve described. We’ll fill in more later in this chapter and in the following

> ones. The next step is turning this grammar into something we can store in memory—syntax trees.

目前， `statement` 只有两种情况，分别对应于我们描述的两类语句。我们将在本章后面和接下来的章节中补充更多内容。接下来就是将这个语法转化为我们可以存储在内存中的东西——语法树。。

> ### #8.1.1 Statement syntax trees

#8.1.1 Statement语法树

> There is no place in the grammar where both an expression and a statement are allowed. The operands of, say, `+` are always expressions, never statements. The body of a `while` loop is always a statement.

语法中没有地方既允许使用表达式，也允许使用语句。 操作符（如`+`）的操作数总是表达式，而不是语句。 `while` 循环的主体总是一个语句。

> Since the two syntaxes are disjoint, we don’t need a single base class that they all inherit from. Splitting expressions and statements into separate class hierarchies enables the Java compiler to help us find dumb mistakes like passing a statement to a Java method that expects an expression.

因为这两种语法是不相干的，所以我们不需要提供一个它们都继承的基类。将表达式和语句拆分为单独的类结构，可使Java编译器帮助我们发现一些愚蠢的错误，例如将语句传递给需要表达式的Java方法。

> That means a new base class for statements. As our elders did before us, we will use the cryptic name “Stmt”. With great foresight, I have designed our little AST metaprogramming script in anticipation of this. That’s why we passed in “Expr” as a parameter to `defineAst()`. Now we add another call to define Stmt and its subclasses.

这意味着要为语句创建一个新的基类。正如我们的前辈那样，我们将使用“Stmt”这个隐秘的名字。我很有远见，在设计我们的AST元编程脚本时就已经预见到了这一点。这就是为什么我们把“Expr”作为参数传给了 `defineAst()` 。现在我们添加另一个方法调用来定义 `Stmt` 和它的子类。

tool/GenerateAst.java，在 main()方法中新增：

```
      "Unary    : Token operator, Expr right"
    ));

    defineAst(outputDir, "Stmt", Arrays.asList(
      "Expression : Expr expression",
      "Print      : Expr expression"
    ));
  }
```

新节点对应的生成代码可以参考附录： Appendix II: Expression statement, Print statement.

> Run the AST generator script and behold the resulting “Stmt.java” file with the syntax tree classes we need for expression and `print` statements. Don’t forget to add the file to your IDE project or makefile or whatever.

运行AST生成器脚本，查看生成的 `Stmt.java` 文件，其中包含表达式和 `print` 语句所需的语法树类。不要忘记将该文件添加到IDE项目或makefile或其他文件中。

> ### #8.1.2 Parsing statements

#8.1.2 解析语句

> The parser’s `parse()` method that parses and returns a single expression was a temporary hack to get the last chapter up and running. Now that our grammar has the correct starting rule, `program`, we can turn `parse()` into the real deal.

解析器的`parse()`方法会解析并返回一个表达式，这是一个临时方案，是为了让上一章的代码能启动并运行起来。现在，我们的语法已经有了正确的起始规则，即`program`，我们可以正式编写`parse()`方法了。

lox/Parser.java，parse()方法，替换7行：

```
  List<Stmt> parse() {
    List<Stmt> statements = new ArrayList<>();
    while (!isAtEnd()) {
      statements.add(statement());
    }

    return statements;
  }
```

> This parses a series of statements, as many as it can find until it hits the end of the input. This is a pretty direct translation of the `program` rule into recursive descent style. We must also chant a minor prayer to the Java verbosity gods since we are using ArrayList now.

该方法会尽可能多地解析一系列语句，直到命中输入内容的结尾为止。这是一种非常直接的将`program`规则转换为递归下降风格的方式。由于我们现在使用ArrayList，所以我们还必须向Java的冗长之神做一个小小的祈祷。

lox/Parser.java，新增代码：

```
package com.craftinginterpreters.lox;
// 新增部分开始
import java.util.ArrayList;
// 新增部分结束
import java.util.List;
```

> A program is a list of statements, and we parse one of those statements using this method:

一个程序就是一系列的语句，而我们可以通过下面的方法解析每一条语句：

lox/Parser.java，在 expression()方法后添加：

```
private Stmt statement() {
  if (match(PRINT)) return printStatement();

  return expressionStatement();
}
```

> A little bare bones, but we'll fill it in with more statement types later. We determine which specific statement rule is matched by looking at the current token. A `print` token means it's obviously a `print` statement.

这是一个简单的框架，但是稍后我们将会填充更多的语句类型。我们通过查看当前标记来确定匹配哪条语句规则。 `print` 标记意味着它显然是一个 `print` 语句。

> If the next token doesn't look like any known kind of statement, we assume it must be an expression statement. That's the typical final fallthrough case when parsing a statement, since it's hard to proactively recognize an expression from its first token.

如果下一个标记看起来不像任何已知类型的语句，我们就认为它一定是一个表达式语句。这是解析语句时典型的最终失败分支，因为我们很难通过第一个标记主动识别出一个表达式。

> Each statement kind gets its own method. First `print`:

每种语句类型都有自己的方法。首先是 `print`：

lox/Parser.java，在 statement()方法后添加：

```
private Stmt printStatement() {
  Expr value = expression();
  consume(SEMICOLON, "Expect ';' after value.");
  return new Stmt.Print(value);
}
```

> Since we already matched and consumed the `print` token itself, we don't need to do that here. We parse the subsequent expression, consume the terminating semicolon, and emit the syntax tree.

因为我们已经匹配并消费了`print`标记本身，所以这里不需要重复消费。我们先解析随后的表达式，消费表示语句终止的分号，并生成语法树。

> If we didn't match a `print` statement, we must have one of these:

如果我们没有匹配到`print`语句，那一定是一条下面的语句：

lox/Parser.java，在 printStatement()方法后添加：

```
private Stmt expressionStatement() {
  Expr expr = expression();
  consume(SEMICOLON, "Expect ';' after expression.");
  return new Stmt.Expression(expr);
}
```

> Similar to the previous method, we parse an expression followed by a semi-colon. We wrap that Expr in a Stmt of the right type and return it.

与前面的方法类似，我们解析一个后面带分号的表达式。我们将Expr封装在一个正确类型的Stmt中，并返回它。

> ### #8.1.3 Executing statements

#8.1.3 执行语句

> We're running through the previous couple of chapters in microcosm, working our way through the front end. Our parser can now produce statement syntax trees, so the next and final step is to interpret them. As in expres-

> sions, we use the Visitor pattern, but we have a new visitor interface, Stmt.Visitor, to implement since statements have their own base class.

我们在前面几章一步一步地慢慢完成了解释器的前端工作。我们的解析器现在可以产生语句语法树，所以下一步，也是最后一步，就是对其进行解释。和表达式一样，我们使用的是Visitor模式，但是我们需要实现一个新的访问者接口`Stmt.Visitor`，因为语句有自己的基类。

> We add that to the list of interfaces Interpreter implements.

我们将其添加到Interpreter实现的接口列表中。

lox/Interpreter.java，替换1行[3]：

```
// 替换部分开始
class Interpreter implements Expr.Visitor<Object>,
                             Stmt.Visitor<Void> {
// 替换部分结束
  void interpret(Expr expression) {
```

> Unlike expressions, statements produce no values, so the return type of the visit methods is Void, not Object. We have two statement types, and we need a visit method for each. The easiest is expression statements.

与表达式不同，语句不会产生值，因此visit方法的返回类型是`Void`，而不是`Object`。我们有两种语句类型，每种类型都需要一个visit方法。最简单的是表达式语句：

lox/Interpreter.java，在 evaluate()方法后添加：

```
  @Override
  public Void visitExpressionStmt(Stmt.Expression stmt) {
    evaluate(stmt.expression);
    return null;
  }
```

> We evaluate the inner expression using our existing `evaluate()` method and discard the value. Then we return `null`. Java requires that to satisfy the special capitalized Void return type. Weird, but what can you do?

我们使用现有的`evaluate()`方法计算内部表达式，并丢弃其结果值。然后我们返回`null`，因为Java要求为特殊的大写Void返回类型返回该值。很奇怪，但你能有什么办法呢？

> The `print` statement's visit method isn't much different.

`print`语句的visit方法没有太大的不同。

lox/Interpreter.java，在 visitExpressionStmt()方法后添加：

```
  @Override
  public Void visitPrintStmt(Stmt.Print stmt) {
    Object value = evaluate(stmt.expression);
    System.out.println(stringify(value));
    return null;
  }
```

> Before discarding the expression's value, we convert it to a string using the `stringify()` method we introduced in the last chapter and then dump it to stdout.

在丢弃表达式的值之前，我们使用上一章引入的`stringify()`方法将其转换为字符串，然后将其输出到stdout。

> Our interpreter is able to visit statements now, but we have some work to do to feed them to it. First, modify the old `interpret()` method in the Interpreter class to accept a list of statements—in other words, a program.

我们的解释器现在可以处理语句了，但是我们还需要做一些工作将语句输入到解释器中。首先，修改Interpreter类中原有的`interpret()`方法，让其能够接受一组语句——即一段程序。

lox/Interpreter.java，修改 interpret()方法，替换8行：

```
  void interpret(List<Stmt> statements) {
    try {
      for (Stmt statement : statements) {
        execute(statement);
      }
    } catch (RuntimeError error) {
      Lox.runtimeError(error);
    }
  }
```

> This replaces the old code which took a single expression. The new code relies on this tiny helper method:

这段代码替换了原先处理单个表达式的旧代码。新代码依赖于下面的小辅助方法。

lox/Interpreter.java，在 evaluate()方法后添加：

```
  private void execute(Stmt stmt) {
    stmt.accept(this);
  }
```

> That’s the statement analogue to the `evaluate()` method we have for expressions. Since we’re working with lists now, we need to let Java know.

这类似于处理表达式的`evaluate()`方法，这是这里处理语句。因为我们要使用列表，所以我们需要在Java中引入一下。

lox/Interpreter.java

```
package com.craftinginterpreters.lox;
// 新增部分开始
import java.util.List;
// 新增部分结束
class Interpreter implements Expr.Visitor<Object>,
```

> The main Lox class is still trying to parse a single expression and pass it to the interpreter. We fix the parsing line like so:

Lox主类中仍然是只解析单个表达式并将其传给解释器。我们将其修正如下：

lox/Lox.java，在 run()方法中替换一行：

```
    Parser parser = new Parser(tokens);
    // 替换部分开始
    List<Stmt> statements = parser.parse();
    // 替换部分结束
    // Stop if there was a syntax error.
```

> And then replace the call to the interpreter with this:

然后将对解释器的调用替换如下：

lox/Lox.java，在 run()方法中替换一行：

```
    if (hadError) return;
    // 替换部分开始
    interpreter.interpret(statements);
    // 替换部分结束
  }
```

> Basically just plumbing the new syntax through. OK, fire up the interpreter and give it a try. At this point, it’s worth sketching out a little Lox program in a text file to run as a script. Something like:

基本就是对新语法进行遍历。 OK，启动解释器并测试一下。 现在有必要在文本文件中草拟一个小的Lox程序来作为脚本运行。 就像是：

```
print "one";
print true;
print 2 + 1;
```

> It almost looks like a real program! Note that the REPL, too, now requires you to enter a full statement instead of a simple expression. Don’t forget your semicolons.

它看起来就像一个真实的程序！ 请注意，REPL现在也要求你输入完整的语句，而不是简单的表达式。 所以不要忘记后面的分号。

> ## #8.2 Global Variables

#8.2 全局变量

> Now that we have statements, we can start working on state. Before we get into all of the complexity of lexical scoping, we’ll start off with the easiest kind of variables—globals. We need two new constructs.

现在我们已经有了语句，可以开始处理状态了。在深入探讨语法作用域的复杂性之前，我们先从最简单的变量（全局变量）开始[4]。我们需要两个新的结构。

1. > A **variable declaration** statement brings a new variable into the world.

   变量声明语句用于创建一个新变量。

   ```
   var beverage = "espresso";
   ```

   > This creates a new binding that associates a name (here “beverage”) with a value (here, the string `"espresso"`).

   该语句将创建一个新的绑定，将一个名称（这里是 `beverage`）和一个值（这里是字符串 `"espresso"`）关联起来。

2. Once that's done, a **variable expression** accesses that binding. When the identifier "beverage" is used as an expression, it looks up the value bound to that name and returns it.

一旦声明完成，**变量表达式**就可以访问该绑定。当标识符“beverage”被用作一个表达式时，程序会查找与该名称绑定的值并返回。

```
print beverage; // "espresso".
```

> Later, we'll add assignment and block scope, but that's enough to get moving.

稍后，我们会添加赋值和块作用域，但是这些已经足够继续后面的学习了。

> ## #8.2.1 Variable syntax

#8.2.1 变量语法

> As before, we'll work through the implementation from front to back, starting with the syntax. Variable declarations are statements, but they are different from other statements, and we're going to split the statement grammar in two to handle them. That's because the grammar restricts where some kinds of statements are allowed.

与前面一样，我们将从语法开始，从前到后依次完成实现。变量声明是一种语句，但它们不同于其他语句，我们把statement语法一分为二来处理该情况。这是因为语法要限制某个位置上哪些类型的语句是被允许的。

> The clauses in control flow statements—think the then and else branches of an `if` statement or the body of a `while`—are each a single statement. But that statement is not allowed to be one that declares a name. This is OK:

控制流语句中的子句——比如，`if`或`while`语句体中的`then`和`else`分支——都是一个语句。但是这个语句不应该是一个声明名称的语句。下面的代码是OK的：

```
if (monday) print "Ugh, already?";
```

> But this is not:

但是下面的代码不行：

```
if (monday) var beverage = "espresso";
```

> We *could* allow the latter, but it's confusing. What is the scope of that `beverage` variable? Does it persist after the `if` statement? If so, what is its value on days other than Monday? Does the variable exist at all on those days?

我们也*可以*允许后者，但是会令人困惑。 `beverage`变量的作用域是什么？ `if`语句结束之后它是否还继续存在？如果存在的话，在其它条件下它的值是什么？这个变量是否在其它情形下也一直存在？

> Code like this is weird, so C, Java, and friends all disallow it. It's as if there are two levels of "precedence" for statements. Some places where a statement is allowed—like inside a block or at the top level—allow any kind of statement, including declarations. Others allow only the "higher" precedence statements that don't declare names.

这样的代码有点奇怪，所以C、Java及类似语言中都不允许这种写法。语句就好像有两个"优先级"。有些允许语句的地方——比如在代码块内或程序顶层[5]——可以允许任何类型的语句，包括变量声明。而其他地方只允许那些不声明名称的、优先级更高的语句。

> To accommodate the distinction, we add another rule for kinds of statements that declare names.

为了适应这种区别，我们为声明名称的语句类型添加了另一条规则：

```
program        → declaration* EOF ;

declaration    → varDecl
               | statement ;

statement      → exprStmt
               | printStmt ;
```

> Declaration statements go under the new `declaration` rule. Right now, it's only variables, but later it will include functions and classes. Any place where a declaration is allowed also allows non-declaring statements, so the `declaration` rule falls through to `statement`. Obviously, you can declare stuff at the top level of a script, so `program` routes to the new rule.

声明语句属于新的 `declaration` 规则。目前，这里只有变量，但是后面还会包含函数和类。任何允许声明的地方都允许一个非声明式的语句，所以 `declaration` 规则会下降到 `statement` 。显然，你可以在脚本的顶层声明一些内容，所以 `program` 规则需要路由到新规则。

> The rule for declaring a variable looks like:

声明一个变量的规则如下：

```
varDecl        → "var" IDENTIFIER ( "=" expression )? ";" ;
```

> Like most statements, it starts with a leading keyword. In this case, `var`. Then an identifier token for the name of the variable being declared, followed by an optional initializer expression. Finally, we put a bow on it with the semicolon.

像大多数语句一样，它以一个前置关键字开头，这里是 `var` 。然后是一个标识符标记，作为声明变量的名称，后面是一个可选的初始化式表达式。最后，以一个分号作为结尾。

> To access a variable, we define a new kind of primary expression.

为了访问变量，我们还需要定义一个新类型的基本表达式：

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

> That `IDENTIFIER` clause matches a single identifier token, which is understood to be the name of the variable being accessed.

`IDENTIFIER` 子语句会匹配单个标识符标记，该标记会被理解为正在访问的变量的名称。

> These new grammar rules get their corresponding syntax trees. Over in the AST generator, we add a new statement tree for a variable declaration.

这些新的语法规则需要其相应的语法树。在AST生成器中，我们为变量声明添加一个新的语句树。

tool/GenerateAst.java，在 main()方法中添加一行，前一行需要加`,`*：*

```
      "Expression : Expr expression",
      "Print      : Expr expression",
      // 新增部分开始
      "Var        : Token name, Expr initializer"
      // 新增部分结束
    ));
```

> It stores the name token so we know what it's declaring, along with the initializer expression. (If there isn't an initializer, that field is `null`.)

这里存储了名称标记，以便我们知道该语句声明了什么，此外还有初始化表达式（如果没有，字段就是`null`）。

> Then we add an expression node for accessing a variable.

然后我们添加一个表达式节点用于访问变量。

tool/GenerateAst.java，在 main()方法中添加一行，前一行需要加`,`*：*

```
      "Literal  : Object value",
      "Unary    : Token operator, Expr right",
      // 新增部分开始
      "Variable : Token name"
      // 新增部分结束
    ));
```

> It's simply a wrapper around the token for the variable name. That's it. As always, don't forget to run the AST generator script so that you get updated "Expr.java" and "Stmt.java" files.

这只是对变量名称标记的简单包装，就是这样。像往常一样，别忘了运行AST生成器脚本，这样你就能得到更新的 "Expr.java "和 "Stmt.java "文件。

> ## #8.2.2 Parsing variables

#8.2.2 解析变量

> Before we parse variable statements, we need to shift around some code to make room for the new `declaration` rule in the grammar. The top level of a program is now a list of declarations, so the entrypoint method to the parser changes.

在解析变量语句之前，我们需要修改一些代码，为语法中的新规则`declaration`腾出一些空间。现在，程序的最顶层是声明语句的列表，所以解析器方法的入口需要更改：

lox/Parser.java，在 parse()方法中替换1行：

```
  List<Stmt> parse() {
    List<Stmt> statements = new ArrayList<>();
    while (!isAtEnd()) {
      // 替换部分开始
      statements.add(declaration());
      // 替换部分结束
    }

    return statements;
  }
```

> That calls this new method:

这里会调用下面的新方法：

lox/Parser.java，在 expression()方法后添加：

```
  private Stmt declaration() {
    try {
      if (match(VAR)) return varDeclaration();

      return statement();
    } catch (ParseError error) {
      synchronize();
      return null;
    }
  }
```

> Hey, do you remember way back in that earlier chapter when we put the infrastructure in place to do error recovery? We are finally ready to hook that up.

你还记得前面的章节中，我们建立了一个进行错误恢复的框架吗？现在我们终于可以用起来了。

> This `declaration()` method is the method we call repeatedly when parsing a series of statements in a block or a script, so it’s the right place to synchro-

> nize when the parser goes into panic mode. The whole body of this method is wrapped in a try block to catch the exception thrown when the parser begins error recovery. This gets it back to trying to parse the beginning of the next statement or declaration.

当我们解析块或脚本中的 一系列语句时， `declaration()` 方法会被重复调用。因此当解析器进入恐慌模式时，它就是进行同步的正确位置。该方法的整个主体都封装在一个try块中，以捕获解析器开始错误恢复时抛出的异常。这样可以让解析器跳转到解析下一个语句或声明的开头。

> The real parsing happens inside the try block. First, it looks to see if we're at a variable declaration by looking for the leading `var` keyword. If not, it falls through to the existing `statement()` method that parses `print` and expression statements.

真正的解析工作发生在try块中。首先，它通过查找前面的 `var` 关键字判断是否是变量声明语句。如果不是的话，就会进入已有的 `statement()` 方法中，解析 `print` 和语句表达式。

> Remember how `statement()` tries to parse an expression statement if no other statement matches? And `expression()` reports a syntax error if it can't parse an expression at the current token? That chain of calls ensures we report an error if a valid declaration or statement isn't parsed.

还记得 `statement()` 会在没有其它语句匹配时会尝试解析一个表达式语句吗？而 `expression()` 如果无法在当前语法标记处解析表达式，则会抛出一个语法错误？这一系列调用链可以保证在解析无效的声明或语句时会报告错误。

> When the parser matches a `var` token, it branches to:

当解析器匹配到一个 `var` 标记时，它会跳转到：

lox/Parser.java，在 printStatement()方法后添加：

```java
  private Stmt varDeclaration() {
    Token name = consume(IDENTIFIER, "Expect variable name.");

    Expr initializer = null;
    if (match(EQUAL)) {
      initializer = expression();
    }

    consume(SEMICOLON, "Expect ';' after variable declaration.");
    return new Stmt.Var(name, initializer);
  }
```

> As always, the recursive descent code follows the grammar rule. The parser has already matched the `var` token, so next it requires and consumes an identifier token for the variable name.

与之前一样，递归下降代码会遵循语法规则。解析器已经匹配了 `var` 标记，所以接下来要消费一个标识符标记作为变量的名称。

> Then, if it sees an `=` token, it knows there is an initializer expression and parses it. Otherwise, it leaves the initializer `null`. Finally, it consumes the required semicolon at the end of the statement. All this gets wrapped in a Stmt.Var syntax tree node and we're groovy.

然后，如果找到 `=` 标记，解析器就知道后面有一个初始化表达式，并对其进行解析。否则，它会将初始器保持为 `null`。最后，会消费语句末尾所需的分号。然后将所有这些都封装到一个Stmt.Var语法树节点中。

> Parsing a variable expression is even easier. In `primary()`, we look for an identifier token.

解析变量表达式甚至更简单。在 `primary()` 中，我们需要查找一个标识符标记。

lox/Parser.java，在 primary()方法中添加：

```
      return new Expr.Literal(previous().literal);
    }
    // 新增部分开始
    if (match(IDENTIFIER)) {
      return new Expr.Variable(previous());
    }
    // 新增部分结束
    if (match(LEFT_PAREN)) {
```

> That gives us a working front end for declaring and using variables. All that’s left is to feed it into the interpreter. Before we get to that, we need to talk about where variables live in memory.

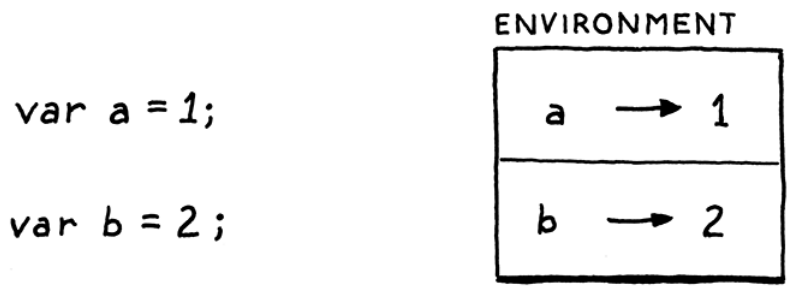
这为我们提供了声明和使用变量的可用前端，剩下的就是将其接入解释器中。在此之前，我们需要讨论变量在内存中的位置。

> ## #8.3 Environments

#8.3 环境

> The bindings that associate variables to values need to be stored somewhere. Ever since the Lisp folks invented parentheses, this data structure has been called an **environment**.

变量与值之间的绑定关系需要保存在某个地方。自从Lisp发明圆括号以来，这种数据结构就被称为**环境**。



> You can think of it like a map where the keys are variable names and the values are the variable's, uh, values. In fact, that's how we'll implement it in Java. We could stuff that map and the code to manage it right into Interpreter, but since it forms a nicely delineated concept, we'll pull it out into its own class.

你可以把它想象成一个映射，其中键是变量名称，值就是变量的值[6]。实际上，这也就是我们在Java中采用的实现方式。我们可以直接在解释器中加入该映射及其管理代码，但是因为它形成了一个很好的概念，我们可以将其提取到单独的类中。

> Start a new file and add:

打开新文件，添加以下代码：

lox/Environment.java，创建新文件

```
package com.craftinginterpreters.lox;

import java.util.HashMap;
import java.util.Map;

class Environment {
  private final Map<String, Object> values = new HashMap<>();
}
```

> There's a Java Map in there to store the bindings. It uses bare strings for the keys, not tokens. A token represents a unit of code at a specific place in the source text, but when it comes to looking up variables, all identifier tokens with the same name should refer to the same variable (ignoring scope for now). Using the raw string ensures all of those tokens refer to the same map key.

其中使用一个Java Map来保存绑定关系。这里使用原生字符串作为键，而不是使用标记。一个标记表示源文本中特定位置的一个代码单元，但是在查找变量

时，具有相同名称的标识符标记都应该指向相同的变量（暂时忽略作用域）。使用原生字符串可以保证所有这些标记都会指向相同的映射键。

> There are two operations we need to support. First, a variable definition binds a new name to a value.

我们需要支持两个操作。首先，是变量定义操作，可以将一个新的名称与一个值进行绑定。

lox/Environment.java，在 Environment类中添加：

```
  void define(String name, Object value) {
    values.put(name, value);
  }
```

> Not exactly brain surgery, but we have made one interesting semantic choice. When we add the key to the map, we don't check to see if it's already present. That means that this program works:

不算困难，但是我们这里也做出了一个有趣的语义抉择。当我们向映射中添加键时，没有检查该键是否已存在。这意味着下面的代码是有效的：

```
var a = "before";
print a; // "before".
var a = "after";
print a; // "after".
```

> A variable statement doesn't just define a *new* variable, it can also be used to *re*define an existing variable. We could choose to make this an error instead. The user may not intend to redefine an existing variable. (If they did mean to, they probably would have used assignment, not `var`.) Making redefinition an error would help them find that bug.

变量语句不仅可以定义一个新变量，也可以用于重新定义一个已有的变量。我们可以选择将其作为一个错误来处理。用户可能不打算重新定义已有的变量

（如果他们想这样做，可能会使用赋值，而不是`var`），将重定义作为错误可以帮助用户发现这个问题。

> However, doing so interacts poorly with the REPL. In the middle of a REPL session, it's nice to not have to mentally track which variables you've already defined. We could allow redefinition in the REPL but not in scripts, but then users would have to learn two sets of rules, and code copied and pasted from one form to the other might not work.

然而，这样做与REPL的交互很差。在与REPL的交互中，最好是让用户不必在脑子记录已经定义了哪些变量。我们可以在REPL中允许重定义，在脚本中不允许。但是这样一来，用户就不得不学习两套规则，而且一种形式的代码复制粘贴到另一种形式后可能无法运行[7]。

> So, to keep the two modes consistent, we'll allow it—at least for global variables. Once a variable exists, we need a way to look it up.

所以，为了保证两种模式的统一，我们选择允许重定义——至少对于全局变量如此。一旦一个变量存在，我们就需要可以查找该变量的方法。

lox/Environment.java，在 Environment类中添加：

```
class Environment {
  private final Map<String, Object> values = new HashMap<>();
  // 新增部分开始
  Object get(Token name) {
    if (values.containsKey(name.lexeme)) {
      return values.get(name.lexeme);
    }

    throw new RuntimeError(name,
        "Undefined variable '" + name.lexeme + "'.");
  }
  // 新增部分结束
  void define(String name, Object value) {
```

> This is a little more semantically interesting. If the variable is found, it simply returns the value bound to it. But what if it's not? Again, we have a choice:

这在语义上更有趣一些。如果找到了这个变量，只需要返回与之绑定的值。但如果没有找到呢？我们又需要做一个选择：

- > Make it a syntax error.

  抛出语法错误

- > Make it a runtime error.

  抛出运行时错误

- > Allow it and return some default value like `nil`.

  允许该操作并返回默认值（如`nil`）

> Lox is pretty lax, but the last option is a little *too* permissive to me. Making it a syntax error—a compile-time error—seems like a smart choice. Using an undefined variable is a bug, and the sooner you detect the mistake, the better.

Lox是很宽松的，但最后一个选项对我来说有点过于宽松了。把它作为语法错误（一个编译时的错误）似乎是一个明智的选择。使用未定义的变量确实是一个错误，用户越早发现这个错误就越好。

> The problem is that *using* a variable isn't the same as *referring* to it. You can refer to a variable in a chunk of code without immediately evaluating it if that chunk of code is wrapped inside a function. If we make it a static error to *mention* a variable before it's been declared, it becomes much harder to define recursive functions.

问题在于，*使用*一个变量并不等同于*引用*它。如果代码块封装在函数中，则可以在代码块中引用变量，而不必立即对其求值。如果我们把引用未声明的变量当作一个静态错误，那么定义递归函数就变得更加困难了。

> We could accommodate single recursion—a function that calls itself—by declaring the function's own name before we examine its body. But that doesn't help with mutually recursive procedures that call each other. Consider:

通过在检查函数体之前先声明函数名称，我们可以支持单一递归——调用自身的函数。但是，这无法处理互相调用的递归程序[8]。考虑以下代码：

```
fun isOdd(n) {
  if (n == 0) return false;
  return isEven(n - 1);
}

fun isEven(n) {
  if (n == 0) return true;
  return isOdd(n - 1);
}
```

> The `isEven()` function isn't defined by the time we are looking at the body of `isOdd()` where it's called. If we swap the order of the two functions, then `isOdd()` isn't defined when we're looking at `isEven()`'s body.

当我们查看`isOdd()`方法时， `isEven()` 方法被调用的时候还没有被声明。如果我们交换着两个函数的顺序，那么在查看`isEven()`方法体时会发现`isOdd()`方法未被定义[9]。

> Since making it a *static* error makes recursive declarations too difficult, we'll defer the error to runtime. It's OK to refer to a variable before it's defined as long as you don't *evaluate* the reference. That lets the program for even and odd numbers work, but you'd get a runtime error in:

因为将其当作*静态错误*会使递归声明过于困难，因此我们把这个错误推迟到运行时。在一个变量被定义之前引用它是可以的，只要你不对引用进行*求值*。这样可以让前面的奇偶数代码正常工作。但是执行以下代码时，你会得到一个运行时错误：

```
print a;
var a = "too late!";
```

> As with type errors in the expression evaluation code, we report a runtime error by throwing an exception. The exception contains the variable’s token so we can tell the user where in their code they messed up.

与表达式计算代码中的类型错误一样，我们通过抛出一个异常来报告运行时错误。异常中包含变量的标记，以便我们告诉用户代码的什么位置出现了错误。

> ### #8.3.1 Interpreting global variables

#8.3.1 解释全局变量

> The Interpreter class gets an instance of the new Environment class.

Interpreter类会获取Environment类的一个实例。

lox/Interpreter.java，在 Interpreter类中添加：

```
class Interpreter implements Expr.Visitor<Object>,
                             Stmt.Visitor<Void> {
  // 添加部分开始
  private Environment environment = new Environment();
  // 添加部分结束
  void interpret(List<Stmt> statements) {
```

> We store it as a field directly in Interpreter so that the variables stay in memory as long as the interpreter is still running.

我们直接将它作为一个字段存储在解释器中，这样，只要解释器仍在运行，变量就会留在内存中。

> We have two new syntax trees, so that’s two new visit methods. The first is for declaration statements.

我们有两个新的语法树，所以这就是两个新的访问方法。第一个是关于声明语句的。

lox/Interpreter.java，在 visitPrintStmt()方法后添加：

```java
@Override
public Void visitVarStmt(Stmt.Var stmt) {
  Object value = null;
  if (stmt.initializer != null) {
    value = evaluate(stmt.initializer);
  }

  environment.define(stmt.name.lexeme, value);
  return null;
}
```

> If the variable has an initializer, we evaluate it. If not, we have another choice to make. We could have made this a syntax error in the parser by *requiring* an initializer. Most languages don’t, though, so it feels a little harsh to do so in Lox.

如果该变量有初始化式，我们就对其求值。如果没有，我们就需要做一个选择。我们可以通过在解析器中*要求*初始化式令其成为一个语法错误。但是，大多数语言都不会这么做，所以在Lox中这样做感觉有点苛刻。

> We could make it a runtime error. We’d let you define an uninitialized variable, but if you accessed it before assigning to it, a runtime error would occur. It’s not a bad idea, but most dynamically typed languages don’t do that. Instead, we’ll keep it simple and say that Lox sets a variable to `nil` if it isn’t explicitly initialized.

我们可以使其成为运行时错误。我们允许您定义一个未初始化的变量，但如果您在对其赋值之前访问它，就会发生运行时错误。这不是一个坏主意，但是大多数动态类型的语言都不会这样做。相反，我们使用最简单的方式。或者说，如果变量没有被显式初始化，Lox会将变量设置为`nil`。

```
var a;
print a; // "nil".
```

> Thus, if there isn't an initializer, we set the value to `null`, which is the Java representation of Lox's `nil` value. Then we tell the environment to bind the variable to that value.

因此，如果没有初始化式，我们将值设为`null`，这也是Lox中的`nil`值的Java表示形式。然后，我们告诉环境上下文将变量与该值进行绑定。

> Next, we evaluate a variable expression.

接下来，我们要对变量表达式求值。

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

```
  @Override
  public Object visitVariableExpr(Expr.Variable expr) {
    return environment.get(expr.name);
  }
```

> This simply forwards to the environment which does the heavy lifting to make sure the variable is defined. With that, we've got rudimentary variables working. Try this out:

这里只是简单地将操作转发到环境上下文中，环境做了一些繁重的工作保证变量已被定义。这样，我们就可以支持基本的变量操作了。尝试以下代码：

```
var a = 1;
var b = 2;
print a + b;
```

> We can't reuse *code* yet, but we can start to build up programs that reuse *data*.

我们还不能复用代码，但是我们可以构建能够复用数据的程序。

> ## #8.4 Assignment

#8.4 赋值

> It's possible to create a language that has variables but does not let you reassign—or **mutate**—them. Haskell is one example. SML supports only mutable references and arrays—variables cannot be reassigned. Rust steers you away from mutation by requiring a `mut` modifier to enable assignment.

你可以创建一种语言，其中有变量，但是不支持对该变量重新赋值（或更改）。Haskell就是一个例子。SML只支持可变引用和数组——变量不能被重新赋值。Rust则通过要求`mut`标识符开启赋值，从而引导用户远离可更改变量。

> Mutating a variable is a side effect and, as the name suggests, some language folks think side effects are dirty or inelegant. Code should be pure math that produces values—crystalline, unchanging ones—like an act of divine creation. Not some grubby automaton that beats blobs of data into shape, one imperative grunt at a time.

更改变量是一种副作用，顾名思义，一些语言专家认为副作用是肮脏或不优雅的。代码应该是纯粹的数学，它会产生值——纯净的、不变的值——就像上帝造物一样。而不是一些肮脏的自动机器，将数据块转换成各种形式，一次执行一条命令。

Lox is not so austere. Lox is an imperative language, and mutation comes with the territory. Adding support for assignment doesn't require much work.

Global variables already support redefinition, so most of the machinery is there now. Mainly, we’re missing an explicit assignment notation.

Lox没有这么严苛。Lox是一个命令式语言，可变性是与生俱来的，添加对赋值操作的支持并不需要太多工作。全局变量已经支持了重定义，所以该机制的大部分功能已经存在。主要的是，我们缺少显式的赋值符号。

> ## #8.4.1 Assignment syntax

#8.4.1 赋值语法

> That little `=` syntax is more complex than it might seem. Like most C-derived languages, assignment is an expression and not a statement. As in C, it is the lowest precedence expression form. That means the rule slots between `expression` and `equality` (the next lowest precedence expression).

这个小小的`=`语法比看起来要更复杂。像大多数C派生语言一样，赋值是一个表达式，而不是一个语句。和C语言中一样，它是优先级最低的表达式形式。这意味着该规则在语法中处于 `expression` 和 `equality` （下一个优先级的表达式）之间。

```
expression     → assignment ;
assignment     → IDENTIFIER "=" assignment
               | equality ;
```

> This says an `assignment` is either an identifier followed by an `=` and an expression for the value, or an `equality` (and thus any other) expression. Later, `assignment` will get more complex when we add property setters on objects, like:

这就是说，一个`assignment` （赋值式）要么是一个标识符，后跟一个`=`和一个对应值的表达式；要么是一个等式（也就是任何其它）表达式。稍后，当我们在对象中添加属性设置式时，赋值将会变得更加复杂，比如：

```
instance.field = "value";
```

> The easy part is adding the new syntax tree node.

最简单的部分就是添加新的语法树节点。

tool/GenerateAst.java，在 main()方法中添加：

```
    defineAst(outputDir, "Expr", Arrays.asList(
      // 新增部分开始
      "Assign   : Token name, Expr value",
      // 新增部分结束
      "Binary   : Expr left, Token operator, Expr right",
```

> It has a token for the variable being assigned to, and an expression for the new value. After you run the AstGenerator to get the new Expr.Assign class, swap out the body of the parser's existing `expression()` method to match the updated rule.

其中包含被赋值变量的标记，一个计算新值的表达式。运行AstGenerator得到新的`Expr.Assign`类之后，替换掉解析器中现有的`expression()`方法的方法体，以匹配最新的规则。

lox/Parser.java，在 expression()方法中替换一行：

```
  private Expr expression() {
    // 替换部分开始
    return assignment();
    // 替换部分结束
  }
```

> Here is where it gets tricky. A single token lookahead recursive descent parser can't see far enough to tell that it's parsing an assignment until *after* it has gone through the left-hand side and stumbled onto the `=`. You might wonder why it even needs to. After all, we don't know we're parsing a `+` expression until after we've finished parsing the left operand.

这里开始变得棘手。单个标记前瞻递归下降解析器直到解析完左侧标记并且遇到 `=` 标记*之后*，才能判断出来正在解析的是赋值语句。你可能会想，为什么需要这样做？毕竟，我们也是完成左操作数的解析之后才知道正在解析的是 `+` 表达式。

> The difference is that the left-hand side of an assignment isn't an expression that evaluates to a value. It's a sort of pseudo-expression that evaluates to a "thing" you can assign to. Consider:

区别在于，赋值表达式的左侧不是可以求值的表达式，而是一种伪表达式，计算出的是一个你可以赋值的"东西"。考虑以下代码：

```
var a = "before";
a = "value";
```

> On the second line, we don't *evaluate* `a` (which would return the string "before"). We figure out what variable `a` refers to so we know where to store the right-hand side expression's value. The classic terms for these two constructs are **l-value** and **r-value**. All of the expressions that we've seen so far that produce values are r-values. An l-value "evaluates" to a storage location that you can assign into.

在第二行中，我们不会对 `a` 进行求值（如果求值会返回"before"）。我们要弄清楚 `a` 指向的是什么变量，这样我们就知道该在哪里保存右侧表达式的值。这两个概念的经典术语是**左值**和**右值**。到目前为止，我们看到的所有产生值的表达式都是右值。左值"计算"会得到一个存储位置，你可以向其赋值。

> We want the syntax tree to reflect that an l-value isn't evaluated like a normal expression. That's why the Expr.Assign node has a *Token* for the left-hand side, not an Expr. The problem is that the parser doesn't know it's parsing an l-value until it hits the `=`. In a complex l-value, that may occur many tokens later.

我们希望语法树能够反映出左值不会像常规表达式那样计算。这也是为什么Expr.Assign节点的左侧是一个Token，而不是Expr。问题在于，解析器直到遇到 `=` 才知道正在解析一个左值。在一个复杂的左值中，可能在出现很多标记之后才能识别到。

```
makeList().head.next = node;
```

> We have only a single token of lookahead, so what do we do? We use a little trick, and it looks like this:

我们只会前瞻一个标记，那我们该怎么办呢？我们使用一个小技巧，看起来像下面这样[10]：

lox/Parser.java，在 expressionStatement()方法后添加：

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

    if (match(EQUAL)) {
      Token equals = previous();
      Expr value = assignment();

      if (expr instanceof Expr.Variable) {
        Token name = ((Expr.Variable)expr).name;
        return new Expr.Assign(name, value);
      }

      error(equals, "Invalid assignment target.");
    }

    return expr;
  }
```

> Most of the code for parsing an assignment expression looks similar to that of the other binary operators like `+`. We parse the left-hand side, which can be any expression of higher precedence. If we find an `=`, we parse the right-hand side and then wrap it all up in an assignment expression tree node.

解析赋值表达式的大部分代码看起来与解析其它二元运算符（如 `+` ）的代码类似。我们解析左边的内容，它可以是任何优先级更高的表达式。如果我们发现一个 `=` ，就解析右侧内容，并把它们封装到一个复杂表达式树节点中。

> One slight difference from binary operators is that we don’t loop to build up a sequence of the same operator. Since assignment is right-associative, we instead recursively call `assignment()` to parse the right-hand side.

与二元运算符的一个细微差别在于，我们不会循环构建相同操作符的序列。因为赋值操作是右关联的，所以我们递归调用 `assignment()` 来解析右侧的值。

> The trick is that right before we create the assignment expression node, we look at the left-hand side expression and figure out what kind of assignment target it is. We convert the r-value expression node into an l-value representation.

诀窍在于，在创建赋值表达式节点之前，我们先查看左边的表达式，弄清楚它是什么类型的赋值目标。然后我们将右值表达式节点转换为左值的表示形式。

> This conversion works because it turns out that every valid assignment target happens to also be valid syntax as a normal expression. Consider a complex field assignment like:

这种转换是有效的，因为事实证明，每个有效的赋值目标正好也是符合普通表达式的有效语法[11]。考虑一个复杂的属性赋值操作，如下：

```
newPoint(x + 2, 0).y = 3;
```

> The left-hand side of that assignment could also work as a valid expression.

该赋值表达式的左侧也是一个有效的表达式。

```
newPoint(x + 2, 0).y;
```

> The first example sets the field, the second gets it.

第一个例子设置该字段，第二个例子获取该字段。

> This means we can parse the left-hand side *as if it were* an expression and then after the fact produce a syntax tree that turns it into an assignment target. If the left-hand side expression isn't a valid assignment target, we fail with a syntax error. That ensures we report an error on code like this:

这意味着，我们可以像解析表达式一样解析左侧内容，然后生成一个语法树，将其转换为赋值目标。如果左边的表达式不是一个有效的赋值目标，就会出现一个语法错误[12]。这样可以确保在遇到类似下面的代码时会报告错误：

```
a + b = c;
```

> Right now, the only valid target is a simple variable expression, but we'll add fields later. The end result of this trick is an assignment expression tree node that knows what it is assigning to and has an expression subtree for the value being assigned. All with only a single token of lookahead and no backtracking.

现在，唯一有效的赋值目标就是一个简单的变量表达式，但是我们后面会添加属性字段。这个技巧的最终结果是一个赋值表达式树节点，该节点知道要向什么赋值，并且有一个表达式子树用于计算要使用的值。所有这些都只用了一个前瞻标记，并且没有回溯。

#8.4.2 Assignment semantics

> We have a new syntax tree node, so our interpreter gets a new visit method.

我们有了一个新的语法树节点，所以我们的解释器也需要一个新的访问方法。

lox/Interpreter.java，在 visitVarStmt()方法后添加：

```java
  @Override
  public Object visitAssignExpr(Expr.Assign expr) {
    Object value = evaluate(expr.value);
    environment.assign(expr.name, value);
    return value;
  }
```

> For obvious reasons, it’s similar to variable declaration. It evaluates the right-hand side to get the value, then stores it in the named variable. Instead of using `define()` on Environment, it calls this new method:

很明显，这与变量声明很类似。首先，对右侧表达式运算以获取值，然后将其保存到命名变量中。这里不使用Environment中的 `define()` ，而是调用下面的新方法：

lox/Environment.java，在 get()方法后添加：

```java
  void assign(Token name, Object value) {
    if (values.containsKey(name.lexeme)) {
      values.put(name.lexeme, value);
      return;
    }

    throw new RuntimeError(name,
        "Undefined variable '" + name.lexeme + "'.");
  }
```

> The key difference between assignment and definition is that assignment is not allowed to create a *new* variable. In terms of our implementation, that means it’s a runtime error if the key doesn’t already exist in the environment’s variable map.

赋值与定义的主要区别在于，赋值操作不允许创建新变量。就我们的实现而言，这意味着如果环境的变量映射中不存在变量的键，那就是一个运行时错误[13]。

> The last thing the `visit()` method does is return the assigned value. That's because assignment is an expression that can be nested inside other expressions, like so:

`visit()`方法做的最后一件事就是返回要赋给变量的值。这是因为赋值是一个表达式，可以嵌套在其他表达式里面，就像这样:

```
var a = 1;
print a = 2; // "2".
```

> Our interpreter can now create, read, and modify variables. It's about as sophisticated as early BASICs. Global variables are simple, but writing a large program when any two chunks of code can accidentally step on each other's state is no fun. We want *local* variables, which means it's time for *scope*.

我们的解释器现在可以创建、读取和修改变量。这和早期的BASICs一样复杂。全局变量很简单，但是在编写一个大型程序时，任何两块代码都可能不小心修改对方的状态，这就不好玩了。我们需要*局部*变量，这意味着是时候讨论*作用域*了。

> ## #8.5 Scope

#8.5 作用域

> A **scope** defines a region where a name maps to a certain entity. Multiple scopes enable the same name to refer to different things in different contexts. In my house, "Bob" usually refers to me. But maybe in your town you know a different Bob. Same name, but different dudes based on where you say it.

作用域定义了名称映射到特定实体的一个区域。多个作用域允许同一个名称在不同的上下文中指向不同的内容。在我家，“Bob”通常指的是我自己，但是在你的身边，你可能认识另外一个Bob。同一个名字，基于你的所知所见指向了不同的人。

> **Lexical scope** (or the less commonly heard **static scope**) is a specific style of scoping where the text of the program itself shows where a scope begins and ends. In Lox, as in most modern languages, variables are lexically scoped. When you see an expression that uses some variable, you can figure out which variable declaration it refers to just by statically reading the code.

词法作用域（或者比较少见的**静态作用域**）是一种特殊的作用域定义方式，程序本身的文本显示了作用域的开始和结束位置[14]。Lox，和大多数现代语言一样，变量在词法作用域内有效。当你看到使用了某些变量的表达式时，你通过静态地阅读代码就可以确定其指向的变量声明。

> For example:

举例来说：

```
{
  var a = "first";
  print a; // "first".
}

{
  var a = "second";
  print a; // "second".
}
```

> Here, we have two blocks with a variable `a` declared in each of them. You and I can tell just from looking at the code that the use of `a` in the first `print` statement refers to the first `a`, and the second one refers to the second.

这里，我们在两个块中都定义了一个变量 `a` 。我们可以从代码中看出，在第一个 `print` 语句中使用的 `a` 指的是第一个 `a` ，第二个语句指向的是第二个变量。

FIRST BLOCK
a
"first"
SECOND BLOCK
a
"second"

> This is in contrast to **dynamic scope** where you don't know what a name refers to until you execute the code. Lox doesn't have dynamically scoped *variables*, but methods and fields on objects are dynamically scoped.

这与**动态作用域**形成了对比，在动态作用域中，直到执行代码时才知道名称指向的是什么。Lox没有动态作用域*变量*，但是对象上的方法和字段是动态作用域的。

```
class Saxophone {
  play() {
    print "Careless Whisper";
  }
}

class GolfClub {
  play() {
    print "Fore!";
  }
}

fun playIt(thing) {
  thing.play();
}
```

When `playIt()` calls `thing.play()` , we don't know if we're about to hear "Careless Whisper" or "Fore!" It depends on whether you pass a Saxophone or a GolfClub to the function, and we don't know that until runtime.

当 `playIt()` 调用 `thing.play()` 时，我们不知道我们将要听到的是 "Careless Whisper "还是 "Fore!" 。这取决于你向函数传递的是Saxophone还是

GolfClub，而我们在运行时才知道这一点。

> Scope and environments are close cousins. The former is the theoretical concept, and the latter is the machinery that implements it. As our interpreter works its way through code, syntax tree nodes that affect scope will change the environment. In a C-ish syntax like Lox's, scope is controlled by curly-braced blocks. (That's why we call it **block scope**.)

作用域和环境是近亲，前者是理论概念，而后者是实现它的机制。当我们的解释器处理代码时，影响作用域的语法树节点会改变环境上下文。在像Lox这样的类C语言语法中，作用域是由花括号的块控制的。（这就是为什么我们称它为**块范围**）。

```
{
  var a = "in block";
}
print a; // Error! No more "a".
```

> The beginning of a block introduces a new local scope, and that scope ends when execution passes the closing `}`. Any variables declared inside the block disappear.

块的开始引入了一个新的局部作用域，当执行通过结束的`}`时，这个作用域就结束了。块内声明的任何变量都会消失。

> ### #8.5.1 Nesting and shadowing

#8.5.1 嵌套和遮蔽

> A first cut at implementing block scope might work like this:

实现块作用域的第一步可能是这样的：

1. > As we visit each statement inside the block, keep track of any variables declared.

   当访问块内的每个语句时，跟踪所有声明的变量。

2. > After the last statement is executed, tell the environment to delete all of those variables.

   执行完最后一条语句后，告诉环境将这些变量全部删除。

> That would work for the previous example. But remember, one motivation for local scope is encapsulation—a block of code in one corner of the program shouldn't interfere with some other block. Check this out:

这对前面的例子是可行的。但是请记住，局部作用域的一个目的是封装——程序中一个块内的代码，不应该干扰其他代码块。看看下面的例子：

```
// How loud?
var volume = 11;

// Silence.
volume = 0;

// Calculate size of 3x4x5 cuboid.
{
  var volume = 3 * 4 * 5;
  print volume;
}
```

> Look at the block where we calculate the volume of the cuboid using a local declaration of `volume`. After the block exits, the interpreter will delete the *global* `volume` variable. That ain't right. When we exit the block, we should remove any variables declared inside the block, but if there is a variable with the same name declared outside of the block, *that's a different variable*. It shouldn't get touched.

请看这个代码块，在这里我们声明了一个局部变量 `volume` 来计算长方体的体积。该代码块退出后，解释器将删除全*局* `volume` 变量。这是不对的。当我们退出代码块时，我们应该删除在块内声明的所有变量，但是如果在代码块外声明了相同名称的变量，那就是一个不同的变量。它不应该被删除。

> When a local variable has the same name as a variable in an enclosing scope, it **shadows** the outer one. Code inside the block can't see it any more—it is hidden in the "shadow" cast by the inner one—but it's still there.

当局部变量与外围作用域中的变量具有相同的名称时，内部变量会遮蔽外部变量。代码块内部不能再看到外部变量——它被遮蔽在内部变量的阴影中——但它仍然是存在的。

> When we enter a new block scope, we need to preserve variables defined in outer scopes so they are still around when we exit the inner block. We do that by defining a fresh environment for each block containing only the variables defined in that scope. When we exit the block, we discard its environment and restore the previous one.

当进入一个新的块作用域时，我们需要保留在外部作用域中定义的变量，这样当我们退出内部代码块时这些外部变量仍然存在。为此，我们为每个代码块定义一个新的环境，该环境只包含该作用域中定义的变量。当我们退出代码块时，我们将丢弃其环境并恢复前一个环境。

> We also need to handle enclosing variables that are *not* shadowed.

我们还需要处理没有被遮蔽的外围变量。

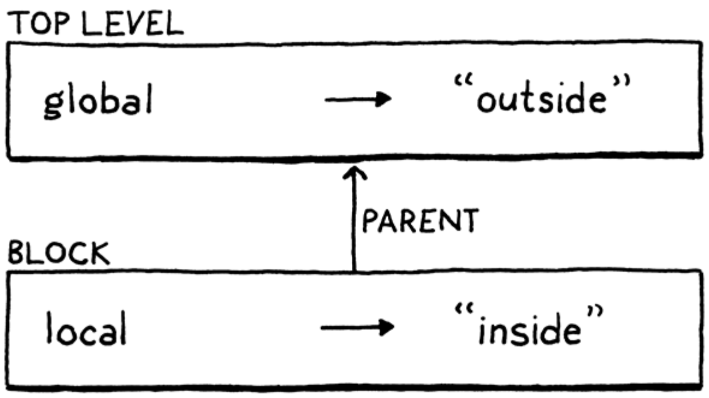
```
var global = "outside";
{
  var local = "inside";
  print global + local;
}
```

> Here, `global` lives in the outer global environment and `local` is defined inside the block's environment. In that `print` statement, both of those variables are in scope. In order to find them, the interpreter must search not only the current innermost environment, but also any enclosing ones.

这段代码中， `global` 在外部全局环境中， `local` 则在块环境中定义。在执行print`语句时，这两个变量都在作用域内。为了找到它们，解释器不仅要搜索当前最内层的环境，还必须搜索所有外围的环境。

> We implement this by chaining the environments together. Each environment has a reference to the environment of the immediately enclosing scope. When we look up a variable, we walk that chain from innermost out until we find the variable. Starting at the inner scope is how we make local variables shadow outer ones.

我们通过将环境链接在一起来实现这一点。每个环境都有一个对直接外围作用域的环境的引用。当我们查找一个变量时，我们从最内层开始遍历环境链直到找到该变量。从内部作用域开始，就是我们使局部变量遮蔽外部变量的方式。



> Before we add block syntax to the grammar, we'll beef up our Environment class with support for this nesting. First, we give each environment a reference to its enclosing one.

在我们添加块语法之前，我们要强化Environment类对这种嵌套的支持。首先，我们在每个环境中添加一个对其外围环境的引用。

lox/Environment.java，在 Environment类中添加：

```
class Environment {
  // 新增部分开始
  final Environment enclosing;
  // 新增部分结束
  private final Map<String, Object> values = new HashMap<>();
```

> This field needs to be initialized, so we add a couple of constructors.

这个字段需要初始化，所以我们添加两个构造函数。

lox/Environment.java，在 Environment类中添加：

```
  Environment() {
    enclosing = null;
  }

  Environment(Environment enclosing) {
    this.enclosing = enclosing;
  }
```

> The no-argument constructor is for the global scope's environment, which ends the chain. The other constructor creates a new local scope nested inside the given outer one.

无参构造函数用于全局作用域环境，它是环境链的结束点。另一个构造函数用来创建一个嵌套在给定外部作用域内的新的局部作用域。

> We don't have to touch the `define()` method—a new variable is always declared in the current innermost scope. But variable lookup and assignment work with existing variables and they need to walk the chain to find them. First, lookup:

我们不必修改`define()`方法——因为新变量总是在当前最内层的作用域中声明。但是变量的查找和赋值是结合已有的变量一起处理的，需要遍历环境链以找到它们。首先是查找操作：

lox/Environment.java，在 get()方法中添加：

```
      return values.get(name.lexeme);
    }
    // 新增部分开始
    if (enclosing != null) return enclosing.get(name);
    // 新增部分结束
    throw new RuntimeError(name,
        "Undefined variable '" + name.lexeme + "'.");
```

> If the variable isn't found in this environment, we simply try the enclosing one. That in turn does the same thing recursively, so this will ultimately walk the entire chain. If we reach an environment with no enclosing one and still don't find the variable, then we give up and report an error as before.

如果当前环境中没有找到变量，就在外围环境中尝试。然后递归地重复该操作，最终会遍历完整个链路。如果我们到达了一个没有外围环境的环境，并且仍然没有找到这个变量，那我们就放弃，并且像之前一样报告一个错误。

> Assignment works the same way.

赋值也是如此。

lox/Environment.java，在 assign()方法中添加：

```
      values.put(name.lexeme, value);
      return;
    }
    // 新增部分开始
    if (enclosing != null) {
      enclosing.assign(name, value);
      return;
    }
    // 新增部分结束
    throw new RuntimeError(name,
```

> Again, if the variable isn't in this environment, it checks the outer one, recursively.

同样，如果变量不在此环境中，它会递归地检查外围环境。

> ## #8.5.2 Block syntax and semantics

#8.5.2 块语法和语义

> Now that Environments nest, we're ready to add blocks to the language. Behold the grammar:

现在环境已经嵌套了，我们就准备向语言中添加块了。请看以下语法：

```
statement      → exprStmt
               | printStmt
               | block ;

block          → "{" declaration* "}" ;
```

> A block is a (possibly empty) series of statements or declarations surrounded by curly braces. A block is itself a statement and can appear anywhere a statement is allowed. The syntax tree node looks like this:

块是由花括号包围的一系列语句或声明(可能是空的)。块本身就是一条语句，可以出现在任何允许语句的地方。语法树节点如下所示。

tool/GenerateAst.java，在 main()方法中添加：

```
    defineAst(outputDir, "Stmt", Arrays.asList(
      // 新增部分开始
      "Block      : List<Stmt> statements",
      // 新增部分结束
      "Expression : Expr expression",
```

> It contains the list of statements that are inside the block. Parsing is straightforward. Like other statements, we detect the beginning of a block by its leading token—in this case the `{`. In the `statement()` method, we add:

它包含块中语句的列表。解析很简单。与其他语句一样，我们通过块的前缀标记(在本例中是`{`)来检测块的开始。在`statement()`方法中，我们添加代码：

lox/Parser.java，在 statement()方法中添加：

```
    if (match(PRINT)) return printStatement();
    // 新增部分开始
    if (match(LEFT_BRACE)) return new Stmt.Block(block());
    // 新增部分结束
    return expressionStatement();
```

> All the real work happens here:

真正的工作都在这里进行：

lox/Parser.java，在 expressionStatement()方法后添加：

```
  private List<Stmt> block() {
    List<Stmt> statements = new ArrayList<>();

    while (!check(RIGHT_BRACE) && !isAtEnd()) {
      statements.add(declaration());
    }

    consume(RIGHT_BRACE, "Expect '}' after block.");
    return statements;
  }
```

> We create an empty list and then parse statements and add them to the list until we reach the end of the block, marked by the closing `}`. Note that the loop also has an explicit check for `isAtEnd()`. We have to be careful to

> avoid infinite loops, even when parsing invalid code. If the user forgets a closing `}`, the parser needs to not get stuck.

我们先创建一个空列表，然后解析语句并将其放入列表中，直至遇到块的结尾（由`}`符号标识）[15]。注意，该循环还有一个明确的`isAtEnd()`检查。我们必须小心避免无限循环，即使在解析无效代码时也是如此。如果用户忘记了结尾的`}`，解析器需要保证不能被阻塞。

> That's it for syntax. For semantics, we add another visit method to Interpreter.

语法到此为止。对于语义，我们要在Interpreter中添加另一个访问方法。

lox/Interpreter.java，在 execute()方法后添加：

```
  @Override
  public Void visitBlockStmt(Stmt.Block stmt) {
    executeBlock(stmt.statements, new Environment(environment));
    return null;
  }
```

> To execute a block, we create a new environment for the block's scope and pass it off to this other method:

要执行一个块，我们先为该块作用域创建一个新的环境，然后将其传入下面这个方法：

lox/Interpreter.java，在execute()方法后添加：

```
  void executeBlock(List<Stmt> statements,
                    Environment environment) {
    Environment previous = this.environment;
    try {
      this.environment = environment;

      for (Stmt statement : statements) {
        execute(statement);
      }
    } finally {
      this.environment = previous;
    }
  }
```

> This new method executes a list of statements in the context of a given environment. Up until now, the `environment` field in Interpreter always pointed to the same environment—the global one. Now, that field represents the *current* environment. That's the environment that corresponds to the innermost scope containing the code to be executed.

这个新方法会在给定的环境上下文中执行一系列语句。在此之前，解释器中的 `environment` 字段总是指向相同的环境——全局环境。现在，这个字段会指向当*前*环境，也就是与要执行的代码的最内层作用域相对应的环境[16]。

> To execute code within a given scope, this method updates the interpreter's `environment` field, visits all of the statements, and then restores the previous value. As is always good practice in Java, it restores the previous environment using a finally clause. That way it gets restored even if an exception is thrown.

为了在给定作用域内执行代码，该方法会先更新解释器的 `environment` 字段，执行所有的语句，然后恢复之前的环境。基于Java中一贯的优良传统，它使用 `finally` 子句来恢复先前的环境。这样一来，即使抛出了异常，环境也会被恢复。

> Surprisingly, that's all we need to do in order to fully support local variables, nesting, and shadowing. Go ahead and try this out:

出乎意料的是，这就是我们为了完全支持局部变量、嵌套和遮蔽所需要做的全部事情。试运行下面的代码：

```
var a = "global a";
var b = "global b";
var c = "global c";
```

```
{
  var a = "outer a";
  var b = "outer b";
  {
    var a = "inner a";
    print a;
    print b;
    print c;
  }
  print a;
  print b;
  print c;
}
print a;
print b;
print c;
```

> Our little interpreter can remember things now. We are inching closer to something resembling a full-featured programming language.

我们的小解释器现在可以记住东西了，我们距离全功能编程语言又近了一步。

```
a = 3;   // OK.
(a) = 3; // Error.
```

> ## #CHALLENGES

#习题

> 1、The REPL no longer supports entering a single expression and automatically printing its result value. That’s a drag. Add support to the REPL to let users type in both statements and expressions. If they enter a state-

> ment, execute it. If they enter an expression, evaluate it and display the result value.

1、REPL不再支持输入一个表达式并自动打印其结果值。这是个累赘。在 REPL 中增加支持，让用户既可以输入语句又可以输入表达式。如果他们输入一个语句，就执行它。如果他们输入一个表达式，则对表达式求值并显示结果值。

> 2、Maybe you want Lox to be a little more explicit about variable initialization. Instead of implicitly initializing variables to `nil`, make it a runtime error to access a variable that has not been initialized or assigned to, as in:

2、也许你希望Lox对变量的初始化更明确一些。与其隐式地将变量初始化为nil，不如将访问一个未被初始化或赋值的变量作为一个运行时错误，如：

```
// No initializers.
var a;
var b;

a = "assigned";
print a; // OK, was assigned first.

print b; // Error!
```

> 3、What does the following program do?

3、下面的代码会怎么执行?

```
var a = 1;
{
  var a = a + 2;
  print a;
}
```

> What did you *expect* it to do? Is it what you think it should do? What does analogous code in other languages you are familiar with do? What do you

> think users will expect this to do?

你期望它怎么执行？它是按照你的想法执行的吗？你所熟悉的其他语言中的类似代码怎么执行？你认为用户会期望它怎么执行？

> ## #DESIGN NOTE: IMPLICIT VARIABLE DECLARATION

#设计笔记：隐式变量声明

> Lox has distinct syntax for declaring a new variable and assigning to an existing one. Some languages collapse those to only assignment syntax. Assigning to a non-existent variable automatically brings it into being. This is called **implicit variable declaration** and exists in Python, Ruby, and CoffeeScript, among others. JavaScript has an explicit syntax to declare variables, but can also create new variables on assignment. Visual Basic has an option to enable or disable implicit variables.
>
> When the same syntax can assign or create a variable, each language must decide what happens when it isn't clear about which behavior the user intends. In particular, each language must choose how implicit declaration interacts with shadowing, and which scope an implicitly declared variable goes into.
>
> - In Python, assignment always creates a variable in the current function's scope, even if there is a variable with the same name declared outside of the function.
> - Ruby avoids some ambiguity by having different naming rules for local and global variables. However, blocks in Ruby (which are more like closures than like "blocks" in C) have their own scope, so it still has the problem. Assignment in Ruby assigns to an existing variable outside of

the current block if there is one with the same name. Otherwise, it creates a new variable in the current block's scope.

- CoffeeScript, which takes after Ruby in many ways, is similar. It explicitly disallows shadowing by saying that assignment always assigns to a variable in an outer scope if there is one, all the way up to the outermost global scope. Otherwise, it creates the variable in the current function scope.
- In JavaScript, assignment modifies an existing variable in any enclosing scope, if found. If not, it implicitly creates a new variable in the *global* scope.

The main advantage to implicit declaration is simplicity. There's less syntax and no "declaration" concept to learn. Users can just start assigning stuff and the language figures it out.

Older, statically typed languages like C benefit from explicit declaration because they give the user a place to tell the compiler what type each variable has and how much storage to allocate for it. In a dynamically typed, garbage-collected language, that isn't really necessary, so you can get away with making declarations implicit. It feels a little more "scripty", more "you know what I mean".

But is that a good idea? Implicit declaration has some problems.

- A user may intend to assign to an existing variable, but may have misspelled it. The interpreter doesn't know that, so it goes ahead and silently creates some new variable and the variable the user wanted to assign to still has its old value. This is particularly heinous in JavaScript where a typo will create a *global* variable, which may in turn interfere with other code.
- JS, Ruby, and CoffeeScript use the presence of an existing variable with the same name—even in an outer scope—to determine whether or not an assignment creates a new variable or assigns to an existing one. That means adding a new variable in a surrounding scope can change the

meaning of existing code. What was once a local variable may silently turn into an assignment to that new outer variable.

- In Python, you may *want* to assign to some variable outside of the current function instead of creating a new variable in the current one, but you can't.

Over time, the languages I know with implicit variable declaration ended up adding more features and complexity to deal with these problems.

- Implicit declaration of global variables in JavaScript is universally considered a mistake today. "Strict mode" disables it and makes it a compile error.
- Python added a `global` statement to let you explicitly assign to a global variable from within a function. Later, as functional programming and nested functions became more popular, they added a similar `nonlocal` statement to assign to variables in enclosing functions.
- Ruby extended its block syntax to allow declaring certain variables to be explicitly local to the block even if the same name exists in an outer scope.

Given those, I think the simplicity argument is mostly lost. There is an argument that implicit declaration is the right *default* but I personally find that less compelling.

My opinion is that implicit declaration made sense in years past when most scripting languages were heavily imperative and code was pretty flat. As programmers have gotten more comfortable with deep nesting, functional programming, and closures, it's become much more common to want access to variables in outer scopes. That makes it more likely that users will run into the tricky cases where it's not clear whether they intend their assignment to create a new variable or reuse a surrounding one.

So I prefer explicitly declaring variables, which is why Lox requires it.

Lox使用不同的语法来声明新变量和为已有变量赋值。有些语言将其简化为只有赋值语法。对一个不存在的变量进行赋值时会自动生成该变量。这被称为**隐式变量声明**，存在于Python、Ruby和CoffeeScript以及其他语言中。JavaScript有一个显式的语法来声明变量，但是也可以在赋值时创建新变量。Visual Basic有一个选项可以启用或禁用隐式变量。

当同样的语法既可以对变量赋值，也可以创建变量时，语言实现就必须决定在不清楚用户的预期行为时该怎么办。特别是，每种语言必须选择隐式变量声明与变量遮蔽的交互方式，以及隐式变量应该属于哪个作用域。

- 在Python中，赋值总是会在当前函数的作用域内创建一个变量，即使在函数外部声明了同名变量。
- Ruby通过对局部变量和全局变量使用不同的命名规则，避免了一些歧义。但是，Ruby中的块（更像闭包，而不是C中的“块”）具有自己的作用域，因此仍然存在问题。在Ruby中，如果已经存在一个同名的变量，则赋值会赋给当前块之外的现有变量。否则，就会在当前块的作用域中创建一个新变量。
- CoffeeScript在许多方面都效仿Ruby，这一点也类似。它明确禁止变量遮蔽，要求赋值时总是优先赋给外部作用域中现有的变量（一直到最外层的全局作用域）。如果变量不存在的话，它会在当前函数作用域中创建新变量。
- 在JavaScript中，赋值会修改任意外部作用域中的一个现有变量（如果能找到该变量的话）。如果变量不存在，它就隐式地在全局作用域内创建一个新的变量。

隐式声明的主要优点是简单。语法较少，无需学习“声明”概念。用户可以直接开始赋值，然后语言就能解决其它问题。

像C这样较早的静态类型语言受益于显式声明，是因为它们给用户提供了一个地方，让他们告诉编译器每个变量的类型以及为它分配多少存储空间。在动态类型、垃圾收集的语言中，这其实是没有必要的，所以你可以通过隐式声明来实现。这感觉更 "脚本化"，更像是 "你懂我的意思吧"。

但这是就个好主意吗？隐式声明还存在一些问题。

- 用户可能打算为现有变量赋值，但是出现拼写错误。解释器不知道这一点，所以它悄悄地创建了一些新变量，而用户想要赋值的变量仍然是原来的值。

这在JavaScript中尤其令人讨厌，因为一个拼写错误会创建一个全局变量，这反过来又可能会干扰其它代码。

- JS、Ruby和CoffeeScript通过判断是否存在同名变量——包括外部作用域——来确定赋值是创建新变量还是赋值给现有变量。这意味着在外围作用域中添加一个新变量可能会改变现有代码的含义，原先的局部变量可能会默默地变成对新的外部变量的赋值。
- 在Python中，你可能想要赋值给当前函数之外的某个变量，而不是在当前函数中创建一个新变量，但是你做不到。

随着时间的推移，我所知道的具有隐式变量声明的语言最后都增加了更多的功能和复杂性来处理这些问题。

- 现在，普遍认为JavaScript中全局变量的隐式声明是一个错误。“Strict mode”禁用了它，并将其成为一个编译错误。
- Python添加了一个 `global` 语句，让用户可以在函数内部显式地赋值给一个全局变量。后来，随着函数式编程和嵌套函数越来越流行，他们添加了一个类似的 `nonlocal` 语句来赋值给外围函数中的变量。
- Ruby扩展了它的块语法，允许在块中显式地声明某些变量，即使外部作用域中存在同名的变量。

考虑到这些，我认为简单性的论点已经失去了意义。有一种观点认为隐式声明是正确的默认选项，但我个人认为这种说法不太有说服力。

我的观点是，隐式声明在过去的几年里是有意义的，当时大多数脚本语言都是非常命令式的，代码是相当简单直观的。随着程序员对深度嵌套、函数式编程和闭包越来越熟悉，访问外部作用域中的变量变得越来越普遍。这使得用户更有可能遇到棘手的情况，即不清楚他们的赋值是要创建一个新变量还是重用外围的已有变量。

所以我更喜欢显式声明变量，这就是Lox要这样做的原因。

---

[^1]

Pascal是一个异类。它区分了过程和函数。函数可以返回值，但过程不能。语言中有一个语句形式用于调用过程，但函数只能在需要表达式的地方被调用。在Pascal中没有表达式语句。

[^2]

我只想说，BASIC和Python有专门的 `print` 语句，而且它们是真正的语言。当然，Python确实在3.0中删除了 `print` 语句。

[^3]

Java不允许使用小写的void作为泛型类型参数，这是因为一些与类型擦除和堆栈有关的隐晦原因。相应的，提供了一个单独的Void类型专门用于此用途，相当于装箱后的void，就像Integer与int的关系。

[^4]

全局状态的名声不好。当然，过多的全局状态（尤其是可变状态）使维护大型程序变得困难。一个出色的软件工程师会尽量减少使用全局变量。但是，如果你正在拼凑一种简单的编程语言，甚至是在学习第一种语言时，全局变量的简单性会有所帮助。我学习的第一门语言是BASIC，虽然我最后不再使用了，但是在我能够熟练使用计算机完成有趣的工作之前，如果能够不需要考虑作用域规则，这一点很好。

[^5]

代码块语句的形式类似于表达式中的括号。“块”本身处于“较高”的优先级，并且可以在任何地方使用，如 `if` 语句的子语句中。而其中*包含的*可以是优先级较低的语句。你可以在块中声明变量或其它名称。通过大括号，你可以在只允许某些语句的位置书写完整的语句语法。

[^6]

Java中称之为**映射**或**哈希映射**。其他语言称它们为**哈希表**、**字典**(Python和c#)、**哈希表**(Ruby和Perl)、**表**(Lua)或**关联数组**(PHP)。很久以前，它们被称为**分散表**。

[^7]

我关于变量和作用域的原则是，“如果有疑问，参考Scheme的做法”。Scheme的开发人员可能比我们花了更多的时间来考虑变量范围的问题——Scheme的主要目标之一就是向世界介绍词法作用域，所以如果你跟随他们的脚步，就很难出错。Scheme允许在顶层重新定义变量。

[^8]

当然，这可能不是判断一个数字是奇偶性的最有效方法（更不用说如果传入一个非整数或负数，程序会发生不可控的事情）。忍耐一下吧。

[^9]

一些静态类型的语言，如Java和C#，通过规定程序的顶层不是一连串的命令式语句来解决这个问题。相应的，它们认为程序是一组同时出现的声明。语言实现在查看任何函数的主体之前，会先声明所有的名字。

像C和Pascal这样的老式语言并不是这样工作的。相反，它们会强制用户添加明确的前向声明，从而在名称完全定义之前先声明它。这是对当时有限的计算能力的一种让步。它们希望能够通过一次文本遍历就编译完一个源文件，因此这些编译器不能在处理函数体之前先收集所有声明。

[^10]

如果左侧不是有效的赋值目标，我们会报告一个错误，但我们不会抛出该错误，因为解析器并没有处于需要进入恐慌模式和同步的混乱状态。

[^11]

即使存在不是有效表达式的赋值目标，你也可以使用这个技巧。定义一个**覆盖语法**，一个可以接受所有有效表达式和赋值目标的宽松语法。如果你遇到了`=`，并且左侧不是有效的赋值目标则报告错误。相对地，如果没有遇到`=`，而且左侧不是有效的表达式也报告一个错误。

[^12]

早在解析一章，我就说过我们要在语法树中表示圆括号表达式，因为我们以后会用到。这就是为什么。我们需要能够区分这些情况：

[^13]

与Python和Ruby不同，Lox不做隐式变量声明。

[^14]

“词法”来自希腊语“ lexikos”，意思是“与单词有关”。 当我们在编程语言中使用它时，通常意味着您无需执行任何操作即可从源代码本身中获取到一些东西。词法作用域是随着ALGOL出现的。早期的语言通常是动态作用域的。当时的计算机科学家认为，动态作用域的执行速度更快。今天，多亏了早期的Scheme研究者，我们知道这不是真的。甚至可以说，情况恰恰相反。变

量的动态作用域仍然存在于某些角落。Emacs Lisp默认为变量的动态作用域。Clojure中的 `binding` 宏也提供了。JavaScript中普遍不被喜欢的 `with` 语句将对象上的属性转换为动态作用域变量。

[^15]

让 `block()` 返回原始的语句列表，并在 `statement()` 方法中将该列表封装在Stmt.Block中，这看起来有点奇怪。我这样做是因为稍后我们会重用 `block()` 来解析函数体，我们当然不希望函数体被封装在Stmt.Block中。

[^16]

手动修改和恢复一个可变的 `environment` 字段感觉很不优雅。另一种经典方法是显式地将环境作为参数传递给每个访问方法。如果要“改变”环境，就在沿树向下递归时传入一个不同的环境。你不必恢复旧的环境，因为新的环境存在于 Java 堆栈中，当解释器从块的访问方法返回时，该环境会被隐式丢弃。我曾考虑过在jlox中这样做，但在每一个访问方法中加入一个环境参数，这有点繁琐和冗长。为了让这本书更简单，我选择了可变字段。