#9.控制流 Control Flow

> Logic, like whiskey, loses its beneficial effect when taken in too large quantities.
>
> —— Edward John Moreton Drax Plunkett, Lord Dunsany

逻辑和威士忌一样，如果摄入太多，就会失去其有益的效果。

> Compared to last chapter’s grueling marathon, today is a lighthearted frolic through a daisy meadow. But while the work is easy, the reward is surprisingly large.

与上一章艰苦的马拉松相比，这一章就是在雏菊草地上的轻松嬉戏。虽然工作很简单，但回报却惊人的大。

> Right now, our interpreter is little more than a calculator. A Lox program can only do a fixed amount of work before completing. To make it run twice as long you have to make the source code twice as lengthy. We’re about to fix that. In this chapter, our interpreter takes a big step towards the programming language major leagues: *Turing-completeness*.

现在，我们的解释器只不过是一个计算器而已。一个Lox程序在结束之前只能做固定的工作量。要想让它的运行时间延长一倍，你就必须让源代码的长度增加一倍。我们即将解决这个问题。在本章中，我们的解释器向编程语言大联盟迈出了一大步：图灵完备性。

#9.1Turing Machines (Briefly)

#9.1 图灵机（简介）

> In the early part of last century, mathematicians stumbled into a series of confusing paradoxes that led them to doubt the stability of the foundation they had built their work upon. To address that crisis, they went back to square one. Starting from a handful of axioms, logic, and set theory, they hoped to rebuild mathematics on top of an impervious foundation.

在上世纪初，数学家们陷入了一系列令人困惑的悖论之中，导致他们对自己工作所依赖的基础的稳定性产生怀疑[1]。为了解决这一危机，他们又回到了原点。他们希望从少量的公理、逻辑和集合理论开始，在一个不透水的地基上重建数学。

> They wanted to rigorously answer questions like, “Can all true statements be proven?”, “Can we compute all functions that we can define?”, or even the more general question, “What do we mean when we claim a function is ‘computable’?”

他们想要严格地回答这样的问题：“所有真实的陈述都可以被证明吗？”，“我们可以计算我们能定义的所有函数吗？”，甚至是更一般性的问题，“当我们声称一个函数是'可计算的'时，代表什么意思？”

> They presumed the answer to the first two questions would be “yes”. All that remained was to prove it. It turns out that the answer to both is “no”, and astonishingly, the two questions are deeply intertwined. This is a fascinating corner of mathematics that touches fundamental questions about what brains are able to do and how the universe works. I can’t do it justice here.

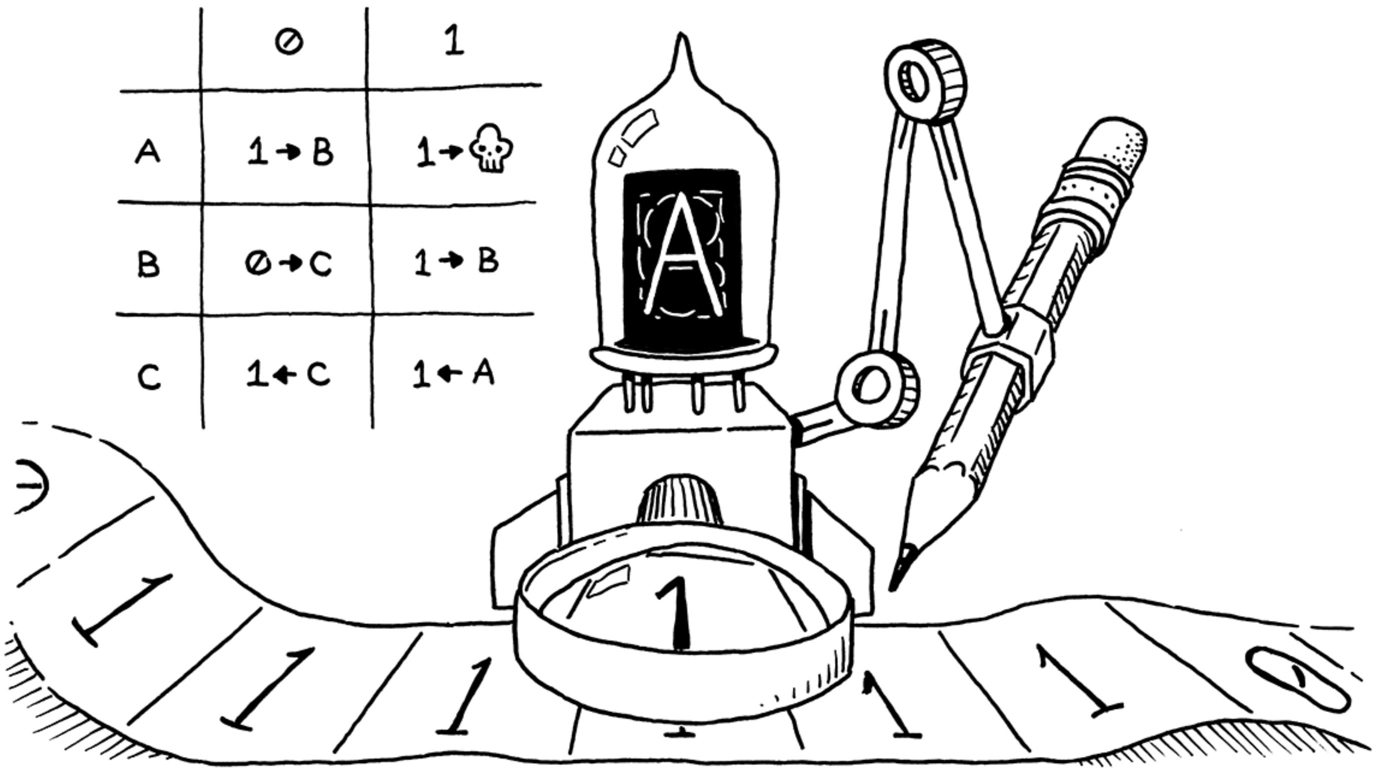
他们认为前两个问题的答案应该是“是”，剩下的就是去证明它。但事实证明这两个问题的答案都是“否”。而且令人惊讶的是，这两个问题是深深地交织在一起的。这是数学的一个迷人的角落，它触及了关于大脑能够做什么和宇宙如何运作的基本问题。我在这里说不清楚。

> What I do want to note is that in the process of proving that the answer to the first two questions is “no”, Alan Turing and Alonzo Church devised a precise answer to the last question—a definition of what kinds of functions are computable. They each crafted a tiny system with a minimum set of machinery that is still powerful enough to compute any of a (very) large class of functions.

我想指出的是，在证明前两个问题的答案是 "否 "的过程中，艾伦·图灵和阿隆佐·邱奇为最后一个问题设计了一个精确的答案，即定义了什么样的函数是可计算的。他们各自设计了一个具有最小机械集的微型系统，该系统仍然强大到足以计算一个超大类函数中的任何一个。

> These are now considered the “computable functions”. Turing’s system is called a **Turing machine**. Church’s is the **lambda calculus**. Both are still widely used as the basis for models of computation and, in fact, many modern functional programming languages use the lambda calculus at their core.

这些现在被认为是“可计算函数”。图灵的系统被称为**图灵机**[2]，邱奇的系统是**lambda演算**。这两种方法仍然被广泛用作计算模型的基础，事实上，许多现代函数式编程语言的核心都是lambda演算。



> Turing machines have better name recognition—there’s no Hollywood film about Alonzo Church yet—but the two formalisms are equivalent in power . In fact, any programming language with some minimal level of expressiveness is powerful enough to compute *any* computable function.

图灵机的知名度更高——目前还没有关于阿隆佐·邱奇的好莱坞电影，但这两种形式在能力上是等价的。事实上，任何具有最低表达能力的编程语言都足以计算任何可计算函数。

> You can prove that by writing a simulator for a Turing machine in your language. Since Turing proved his machine can compute any computable function, by extension, that means your language can too. All you need to do is translate the function into a Turing machine, and then run that on your simulator.

你可以用自己的语言为图灵机编写一个模拟器来证明这一点。由于图灵证明了他的机器可以计算任何可计算函数，推而广之，这意味着你的语言也可以。你所需要做的就是把函数翻译成图灵机，然后在你的模拟器上运行它。

> If your language is expressive enough to do that, it’s considered **Turing-complete**. Turing machines are pretty dang simple, so it doesn’t take much power to do this. You basically need arithmetic, a little control flow, and the ability to allocate and use (theoretically) arbitrary amounts of memory. We’ve got the first. By the end of this chapter, we’ll have the second.

如果你的语言有足够的表达能力来做到这一点，它就被认为是**图灵完备**的。图灵机非常简单，所以它不需要太多的能力。您基本上只需要算术、一点控制流以及分配和使用(理论上)任意数量内存的能力。我们已经具备了第一个条件[3]。在本章结束时，我们将具备第二个条件。

> ## #9.2Conditional Execution

#9.2 条件执行

> Enough history, let’s jazz up our language. We can divide control flow roughly into two kinds:

说完了历史，现在让我们把语言优化一下。我们大致可以把控制流分为两类：

- > **Conditional** or **branching control flow** is used to *not* execute some piece of code. Imperatively, you can think of it as jumping *ahead* over a region of code.

  条件或**分支控制流**是用来不执行某些代码的。意思是，你可以把它看作是跳过了代码的一个区域。

- > **Looping control flow** executes a chunk of code more than once. It jumps *back* so that you can do something again. Since you don’t usually

> want *infinite* loops, it typically has some conditional logic to know when to stop looping as well.

循环控制流是用于多次执行一块代码的。它会*向回*跳转，从而能再次执行某些代码。用户通常不需要无限循环，所以一般也会有一些条件逻辑用于判断何时停止循环。

> Branching is simpler, so we'll start there. C-derived languages have two main conditional execution features, the `if` statement and the perspicaciously named "conditional" operator (`?:`). An `if` statement lets you conditionally execute statements and the conditional operator lets you conditionally execute expressions.

分支更简单一些，所以我们先从分支开始实现。C衍生语言中包含两个主要的条件执行功能，即`if`语句和"条件"运算符（`?:`）[4]。`if`语句使你可以按条件执行语句，而条件运算符使你可以按条件执行表达式。

> For simplicity's sake, Lox doesn't have a conditional operator, so let's get our `if` statement on. Our statement grammar gets a new production.

为了简单起见，Lox没有条件运算符，所以让我们直接开始`if`语句吧。我们的语句语法需要一个新的生成式。

```
statement      → exprStmt
               | ifStmt
               | printStmt
               | block ;

ifStmt         → "if" "(" expression ")" statement
               ( "else" statement )? ;
```

> An `if` statement has an expression for the condition, then a statement to execute if the condition is truthy. Optionally, it may also have an `else` keyword and a statement to execute if the condition is falsey. The syntax tree node has fields for each of those three pieces.

if语句有一个表达式作为条件，然后是一个在条件为真时要执行的语句。另外，它还可以有一个 `else` 关键字和条件为假时要执行的语句。语法树节点中对语法的这三部分都有对应的字段。

tool/GenerateAst.java，在 main()方法中添加：

```
      "Expression : Expr expression",
      // 新增部分开始
      "If         : Expr condition, Stmt thenBranch," +
                  " Stmt elseBranch",
      // 新增部分结束
      "Print      : Expr expression",
```

> Like other statements, the parser recognizes an `if` statement by the leading `if` keyword.

与其它语句类似，解析器通过开头的 `if` 关键字来识别 `if` 语句。

lox/Parser.java，在 statement()方法中添加：

```
  private Stmt statement() {
    // 新增部分开始
    if (match(IF)) return ifStatement();
    // 新增部分结束
    if (match(PRINT)) return printStatement();
```

> When it finds one, it calls this new method to parse the rest:

如果发现了 `if` 关键字，就调用下面的新方法解析其余部分[5]：

lox/Parser.java，在 statement()方法后添加：

```
  private Stmt ifStatement() {
    consume(LEFT_PAREN, "Expect '(' after 'if'.");
    Expr condition = expression();
    consume(RIGHT_PAREN, "Expect ')' after if condition.");

    Stmt thenBranch = statement();
    Stmt elseBranch = null;
    if (match(ELSE)) {
      elseBranch = statement();
    }

    return new Stmt.If(condition, thenBranch, elseBranch);
  }
```

> As usual, the parsing code hews closely to the grammar. It detects an else clause by looking for the preceding `else` keyword. If there isn't one, the `elseBranch` field in the syntax tree is `null`.

跟之前一样，解析代码严格遵循语法。它通过查找前面的`else`关键字来检测`else`子句。如果没有，语法树中的`elseBranch`字段为`null`。

> That seemingly innocuous optional else has, in fact, opened up an ambiguity in our grammar. Consider:

实际上，这个看似无伤大雅的可选项在我们的语法中造成了歧义。考虑以下代码：

```
if (first) if (second) whenTrue(); else whenFalse();
```

> Here's the riddle: Which `if` statement does that else clause belong to? This isn't just a theoretical question about how we notate our grammar. It actually affects how the code executes:

谜题是这样的:这里的`else`子句属于哪个`if`语句?这不仅仅是一个关于如何标注语法的理论问题。它实际上会影响代码的执行方式：

- > If we attach the else to the first `if` statement, then `whenFalse()` is called if `first` is falsey, regardless of what value `second` has.

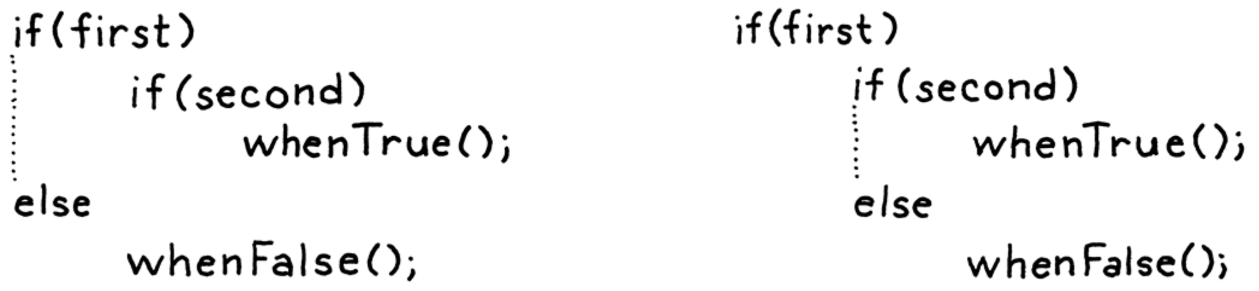
  如果我们将`else`语句关联到第一个`if`语句，那么当`first`为假时，无论`second`的值是多少，都将调用`whenFalse()`。

- > If we attach it to the second `if` statement, then `whenFalse()` is only called if `first` is truthy and `second` is falsey.

  如果我们将`else`语句关联到第二个`if`语句，那么只有当`first`为真并且`second`为假时，才会调用`whenFalse()`。

> Since else clauses are optional, and there is no explicit delimiter marking the end of the `if` statement, the grammar is ambiguous when you nest `if`s in this way. This classic pitfall of syntax is called the **dangling else** problem.

由于`else`子句是可选的，而且没有明确的分隔符来标记`if`语句的结尾，所以当你以这种方式嵌套`if`时，语法是不明确的。这种典型的语法陷阱被称为悬空的else问题。



> It *is* possible to define a context-free grammar that avoids the ambiguity directly, but it requires splitting most of the statement rules into pairs, one that allows an `if` with an `else` and one that doesn't. It's annoying.

也可以定义一个上下文无关的语法来直接避免歧义，但是需要将大部分语句规则拆分成对，一个是允许带有`else`的`if`语句，另一个不允许。这很烦人。

> Instead, most languages and parsers avoid the problem in an ad hoc way. No matter what hack they use to get themselves out of the trouble, they always choose the same interpretation—the `else` is bound to the nearest `if` that precedes it.

相反，大多数语言和解析器都以一种特殊的方式避免了这个问题。不管他们用什么方法来解决这个问题，他们总是选择同样的解释—— `else` 与前面最近的 `if` 绑定在一起。

> Our parser conveniently does that already. Since `ifStatement()` eagerly looks for an `else` before returning, the innermost call to a nested series will claim the else clause for itself before returning to the outer `if` statements.

我们的解析器已经很方便地做到了这一点。因为 `ifStatement()` 在返回之前会继续寻找一个 `else` 子句，连续嵌套的最内层调用在返回外部的 `if` 语句之前，会先为自己声明 `else` 语句。

> Syntax in hand, we are ready to interpret.

语法就绪了，我们可以开始解释了。

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

```
  @Override
  public Void visitIfStmt(Stmt.If stmt) {
    if (isTruthy(evaluate(stmt.condition))) {
      execute(stmt.thenBranch);
    } else if (stmt.elseBranch != null) {
      execute(stmt.elseBranch);
    }
    return null;
  }
```

> The interpreter implementation is a thin wrapper around the self-same Java code. It evaluates the condition. If truthy, it executes the then branch. Otherwise, if there is an else branch, it executes that.

解释器实现就是对相同的Java代码的简单包装。它首先对条件表达式进行求值。如果为真，则执行 `then` 分支。否则，如果有存在 `else` 分支，就执行该分支。

> If you compare this code to how the interpreter handles other syntax we've implemented, the part that makes control flow special is that Java `if` statement. Most other syntax trees always evaluate their subtrees. Here, we may not evaluate the then or else statement. If either of those has a side effect, the choice not to evaluate it becomes user visible.

如果你把这段代码与解释器中我们已实现的处理其它语法的代码进行比较，会发现控制流中特殊的地方就在于Java的 `if` 语句。其它大多数语法树总是会对子树求值，但是这里，我们可能会不执行 `then` 语句或 `else` 语句。如果其中任何一个语句有副作用，那么选择不执行某条语句就是用户可见的。

> ## #9.3Logical Operators

#9.3 逻辑操作符

> Since we don't have the conditional operator, you might think we're done with branching, but no. Even without the ternary operator, there are two other operators that are technically control flow constructs—the logical operators `and` and `or` .

由于我们没有条件运算符，你可能认为我们已经完成分支开发了，但其实还没有。虽然没有三元运算符，但是还有两个其它操作符在技术上是控制流结构——逻辑运算符 `and` 和 `or` 。

> These aren't like other binary operators because they **short-circuit**. If, after evaluating the left operand, we know what the result of the logical expression must be, we don't evaluate the right operand. For example:

它们与其它二进制运算符不同，是因为它们会短路。如果在计算左操作数之后，我们已经确切知道逻辑表达式的结果，那么就不再计算右操作数。例如：

```
false and sideEffect();
```

> For an `and` expression to evaluate to something truthy, both operands must be truthy. We can see as soon as we evaluate the left `false` operand that that isn't going to be the case, so there's no need to evaluate `sideEffect()` and it gets skipped.

对于一个 `and` 表达式来说，两个操作数都必须是真，才能得到结果为真。我们只要看到左侧的 `false` 操作数，就知道结果不会是真，也就不需要对 `sideEffect()` 求值，会直接跳过它。

> This is why we didn't implement the logical operators with the other binary operators. Now we're ready. The two new operators are low in the precedence table. Similar to `||` and `&&` in C, they each have their own precedence with `or` lower than `and`. We slot them right between `assignment` and `equality`.

这就是为什么我们没有在实现其它二元运算符的时候一起实现逻辑运算符。现在我们已经准备好了。这两个新的运算符在优先级表中的位置很低，类似于C语言中的 `||` 和 `&&`，它们都有各自的优先级， `or` 低于 `and` 。我们把这两个运算符插入 `assignment` 和 `equality` 之间。

```
expression     → assignment ;
assignment     → IDENTIFIER "=" assignment
               | logic_or ;
logic_or       → logic_and ( "or" logic_and )* ;
logic_and      → equality ( "and" equality )* ;
```

> Instead of falling back to `equality`, `assignment` now cascades to `logic_or`. The two new rules, `logic_or` and `logic_and`, are similar to other binary operators. Then `logic_and` calls out to `equality` for its operands, and we chain back to the rest of the expression rules.

`assignment` 现在不是落到 `equality`，而是继续进入 `logic_or`。两个新规则，`logic_or` 和 `logic_and`，与其它二元运算符类似。然后 `logic_and` 会调用 `equality` 计算其操作数，然后我们就链入了表达式规则的其它部分。

> We could reuse the existing Expr.Binary class for these two new expressions since they have the same fields. But then `visitBinaryExpr()` would have to check to see if the operator is one of the logical operators and use a different code path to handle the short circuiting. I think it's cleaner to define a new class for these operators so that they get their own visit method.

对于这两个新表达式，我们可以重用Expr.Binary类，因为他们具有相同的字段。但是这样的话，`visitBinaryExpr()` 方法中必须检查运算符是否是逻辑运算符，并且要使用不同的代码处理短路逻辑。我认为更整洁的方法是为这些运算符定义一个新类，这样它们就有了自己的 `visit` 方法。

tool/GenerateAst.java，在main()方法中添加：

```
      "Literal  : Object value",
      // 新增部分开始
      "Logical  : Expr left, Token operator, Expr right",
      // 新增部分结束
      "Unary    : Token operator, Expr right",
```

> To weave the new expressions into the parser, we first change the parsing code for assignment to call `or()`.

为了将新的表达式加入到解析器中，我们首先将赋值操作的解析代码改为调用 `or()` 方法。

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

```
  private Expr assignment() {
    // 新增部分开始
    Expr expr = or();
    // 新增部分结束
    if (match(EQUAL)) {
```

> The code to parse a series of `or` expressions mirrors other binary operators.

解析一系列 `or` 语句的代码与其它二元运算符相似。

lox/Parser.java，在 assignment()方法后添加：

```
private Expr or() {
  Expr expr = and();

  while (match(OR)) {
    Token operator = previous();
    Expr right = and();
    expr = new Expr.Logical(expr, operator, right);
  }

  return expr;
}
```

> Its operands are the next higher level of precedence, the new `and` expression.

它的操作数是位于下一优先级的新的 `and` 表达式。

lox/Parser.java，在 or()方法后添加：

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

  while (match(AND)) {
    Token operator = previous();
    Expr right = equality();
    expr = new Expr.Logical(expr, operator, right);
  }

  return expr;
}
```

> That calls `equality()` for its operands, and with that, the expression parser is all tied back together again. We're ready to interpret.

这里会调用 `equality()` 计算操作数，这样一来，表达式解析器又重新绑定到了一起。我们已经准备好进行解释了。

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

```java
@Override
public Object visitLogicalExpr(Expr.Logical expr) {
  Object left = evaluate(expr.left);

  if (expr.operator.type == TokenType.OR) {
    if (isTruthy(left)) return left;
  } else {
    if (!isTruthy(left)) return left;
  }

  return evaluate(expr.right);
}
```

> If you compare this to the earlier chapter's `visitBinaryExpr()` method, you can see the difference. Here, we evaluate the left operand first. We look at its value to see if we can short-circuit. If not, and only then, do we evaluate the right operand.

如果你把这个方法与前面章节的 `visitBinaryExpr()` 方法相比较，就可以看出其中的区别。这里，我们先计算左操作数。然后我们查看结果值，判断是否可以短路。当且仅当不能短路时，我们才计算右侧的操作数。

> The other interesting piece here is deciding what actual value to return. Since Lox is dynamically typed, we allow operands of any type and use truthiness to determine what each operand represents. We apply similar reasoning to the result. Instead of promising to literally return `true` or `false`, a logic operator merely guarantees it will return a value with appropriate truthiness.

另一个有趣的部分是决定返回什么实际值。由于Lox是动态类型的，我们允许任何类型的操作数，并使用真实性来确定每个操作数代表什么。我们对结果采

用类似的推理。逻辑运算符并不承诺会真正返回`true`或`false`，而只是保证它将返回一个具有适当真实性的值。

> Fortunately, we have values with proper truthiness right at hand—the results of the operands themselves. So we use those. For example:

幸运的是，我们手边就有具有适当真实性的值——即操作数本身的结果，所以我们可以直接使用它们。如:

```
print "hi" or 2; // "hi".
print nil or "yes"; // "yes".
```

> On the first line, `"hi"` is truthy, so the `or` short-circuits and returns that. On the second line, `nil` is falsey, so it evaluates and returns the second operand, `"yes"`.

在第一行，`“hi”`是真的，所以`or`短路并返回它。在第二行，`nil`是假的，因此它计算并返回第二个操作数`“yes”`。

> That covers all of the branching primitives in Lox. We're ready to jump ahead to loops. You see what I did there? *Jump. Ahead.* Get it? See, it's like a reference to . . . oh, forget it.

这样就完成了Lox中的所有分支原语，我们准备实现循环吧。

> ## #9.4While Loops

#9.4 While循环

> Lox features two looping control flow statements, `while` and `for`. The `while` loop is the simpler one, so we'll start there. Its grammar is the same as in C.

Lox有两种类型的循环控制流语句，分别是`while`和`for`。`while`循环更简单一点，我们先从它开始.

```
statement      → exprStmt
               | ifStmt
               | printStmt
               | whileStmt
               | block ;

whileStmt      → "while" "(" expression ")" statement ;
```

> We add another clause to the statement rule that points to the new rule for while. It takes a `while` keyword, followed by a parenthesized condition expression, then a statement for the body. That new grammar rule gets a syntax tree node.

我们在`statement`规则中添加一个子句，指向while对应的新规则`whileStmt`。该规则接收一个`while`关键字，后跟一个带括号的条件表达式，然后是循环体对应的语句。新语法规则需要定义新的语法树节点。

tool/GenerateAst.java,在 main()方法中新增，前一行后添加“,”

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

> The node stores the condition and body. Here you can see why it's nice to have separate base classes for expressions and statements. The field declarations make it clear that the condition is an expression and the body is a statement.

该节点中保存了条件式和循环体。这里就可以看出来为什么表达式和语句最好要有单独的基类。字段声明清楚地表明了，条件是一个表达式，循环主体是一个语句。

> Over in the parser, we follow the same process we used for `if` statements. First, we add another case in `statement()` to detect and match the leading keyword.

在解析器中，我们遵循与`if`语句相同的处理步骤。首先，在 `statement()` 添加一个case分支检查并匹配开头的关键字。

lox/Parser.java，在statement()方法中添加：

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

> That delegates the real work to this method:

实际的工作委托给下面的方法：

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

```
  private Stmt whileStatement() {
    consume(LEFT_PAREN, "Expect '(' after 'while'.");
    Expr condition = expression();
    consume(RIGHT_PAREN, "Expect ')' after condition.");
    Stmt body = statement();

    return new Stmt.While(condition, body);
  }
```

> The grammar is dead simple and this is a straight translation of it to Java. Speaking of translating straight to Java, here’s how we execute the new syntax:

语法非常简单，这里将其直接翻译为Java。说到直接翻译成Java，下面是我们执行新语法的方式：

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

```java
  @Override
  public Void visitWhileStmt(Stmt.While stmt) {
    while (isTruthy(evaluate(stmt.condition))) {
      execute(stmt.body);
    }
    return null;
  }
```

> Like the visit method for `if`, this visitor uses the corresponding Java feature. This method isn't complex, but it makes Lox much more powerful. We can finally write a program whose running time isn't strictly bound by the length of the source code.

和`if`的访问方法一样，这里的访问方法使用了相应的Java特性。这个方法并不复杂，但它使Lox变得更加强大。我们终于可以编写一个运行时间不受源代码长度严格限制的程序了。

> ## #9.5For Loops

#9.5 For循环

> We're down to the last control flow construct, Ye Olde C-style `for` loop. I probably don't need to remind you, but it looks like this:

我们已经到了最后一个控制流结构，即老式的C语言风格`for`循环。我可能不需要提醒你，但还是要说它看起来是这样的：

```
for (var i = 0; i < 10; i = i + 1) print i;
```

> In grammarese, that's:

在语法中，是这样的：

```
statement      → exprStmt
               | forStmt
               | ifStmt
               | printStmt
               | whileStmt
               | block ;

forStmt        → "for" "(" ( varDecl | exprStmt | ";" )
                 expression? ";"
                 expression? ")" statement ;
```

> Inside the parentheses, you have three clauses separated by semicolons:

在括号内，有三个由分号分隔的子语句：

1. > The first clause is the *initializer*. It is executed exactly once, before anything else. It's usually an expression, but for convenience, we also allow a variable declaration. In that case, the variable is scoped to the rest of the `for` loop—the other two clauses and the body.

   第一个子句是*初始化式*。它只会在任何其它操作之前执行一次。它通常是一个表达式，但是为了便利，我们也允许一个变量声明。在这种情况下，变量的作用域就是 `for` 循环的其它部分——其余两个子式和循环体。

2. > Next is the *condition*. As in a `while` loop, this expression controls when to exit the loop. It's evaluated once at the beginning of each iteration, including the first. If the result is truthy, it executes the loop body. Otherwise, it bails.

   接下来是*条件表达式*。与 `while` 循环一样，这个表达式控制了何时退出循环。它会在每次循环开始之前执行一次（包括第一次）。如果结果是真，就执行循环体；否则，就结束循环。

3. > The last clause is the *increment*. It's an arbitrary expression that does some work at the end of each loop iteration. The result of the expression

> is discarded, so it must have a side effect to be useful. In practice, it usually increments a variable.

最后一个子句是*增量式*。它是一个任意的表达式，会在每次循环结束的时候做一些工作。因为表达式的结果会被丢弃，所以它必须有副作用才能有用。在实践中，它通常会对变量进行递增。

> Any of these clauses can be omitted. Following the closing parenthesis is a statement for the body, which is typically a block.

这些子语句都可以忽略。在右括号之后是一个语句作为循环体，通常是一个代码块。

> #9.5.1Desugaring

#9.5.1 语法脱糖

> That’s a lot of machinery, but note that none of it does anything you couldn’t do with the statements we already have. If `for` loops didn’t support initializer clauses, you could just put the initializer expression before the `for` statement. Without an increment clause, you could simply put the increment expression at the end of the body yourself.

这里包含了很多配件，但是请注意，它所做的任何事情中，没有一件是无法用已有的语句实现的。如果 `for` 循环不支持初始化子句，你可以在 `for` 语句之前加一条初始化表达式。如果没有增量子语句，你可以直接把增量表达式放在循环体的最后。

> In other words, Lox doesn’t *need* `for` loops, they just make some common code patterns more pleasant to write. These kinds of features are called **syntactic sugar**. For example, the previous `for` loop could be rewritten like so:

换句话说，Lox不*需要*`for`循环，它们只是让一些常见的代码模式更容易编写。这类功能被称为**语法糖**[6]。例如，前面的`for`循环可以改写成这样：

```
{
  var i = 0;
  while (i < 10) {
    print i;
    i = i + 1;
  }
}
```

> This script has the exact same semantics as the previous one, though it’s not as easy on the eyes. Syntactic sugar features like Lox’s `for` loop make a language more pleasant and productive to work in. But, especially in sophisticated language implementations, every language feature that requires back-end support and optimization is expensive.

虽然这个脚本不太容易看懂，但这个脚本与之前那个语义完全相同。像Lox中的`for`循环这样的语法糖特性可以使语言编写起来更加愉快和高效。但是，特别是在复杂的语言实现中，每一个需要后端支持和优化的语言特性都是代价昂贵的。

> We can have our cake and eat it too by **desugaring**. That funny word describes a process where the front end takes code using syntax sugar and translates it to a more primitive form that the back end already knows how to execute.

我们可以通过**脱糖**来吃这个蛋糕。这个有趣的词描述了这样一个过程：前端接收使用了语法糖的代码，并将其转换成后端知道如何执行的更原始的形式。

> We’re going to desugar `for` loops to the `while` loops and other statements the interpreter already handles. In our simple interpreter, desugaring really doesn’t save us much work, but it does give me an excuse to introduce you to the technique. So, unlike the previous statements, we *won’t* add a new

> syntax tree node. Instead, we go straight to parsing. First, add an import we'll need soon.

我们将把`for`循环脱糖为`while`循环和其它解释器可处理的其它语句。在我们的简单解释器中，脱糖真的不能为我们节省很多工作，但它确实给了我一个契机来向你介绍这一技术。因此，与之前的语句不同，我们不会为`for`循环添加一个新的语法树节点。相反，我们会直接进行解析。首先，先引入一个我们要用到的依赖：

lox/Parser.java，添加代码：

```
import java.util.ArrayList;
// 新增部分开始
import java.util.Arrays;
// 新增部分结束
import java.util.List;
```

> Like every statement, we start parsing a `for` loop by matching its keyword.

像每个语句一样，我们通过匹配`for`关键字来解析循环。

lox/Parser.java,在 statement()方法中新增：

```
  private Stmt statement() {
    // 新增部分开始
    if (match(FOR)) return forStatement();
    // 新增部分结束
    if (match(IF)) return ifStatement();
```

> Here is where it gets interesting. The desugaring is going to happen here, so we'll build this method a piece at a time, starting with the opening parenthesis before the clauses.

接下来是有趣的部分，脱糖也是在这里发生的，所以我们会一点点构建这个方法，首先从子句之前的左括号开始。

lox/Parser.java，在 statement()方法后添加：

```
  private Stmt forStatement() {
    consume(LEFT_PAREN, "Expect '(' after 'for'.");

    // More here...
  }
```

> The first clause following that is the initializer.

接下来的第一个子句是初始化式。

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

```
    consume(LEFT_PAREN, "Expect '(' after 'for'.");
    // 替换部分开始
    Stmt initializer;
    if (match(SEMICOLON)) {
      initializer = null;
    } else if (match(VAR)) {
      initializer = varDeclaration();
    } else {
      initializer = expressionStatement();
    }
    // 替换部分结束
  }
```

> If the token following the `(` is a semicolon then the initializer has been omitted. Otherwise, we check for a `var` keyword to see if it's a variable declaration. If neither of those matched, it must be an expression. We parse that and wrap it in an expression statement so that the initializer is always of type Stmt.

如果`(`后面的标记是分号，那么初始化式就被省略了。否则，我们就检查`var`关键字，看它是否是一个变量声明。如果这两者都不符合，那么它一定是一个表达式。我们对其进行解析，并将其封装在一个表达式语句中，这样初始化器就必定属于Stmt类型。

> Next up is the condition.

接下来是条件表达式。

lox/Parser.java，在 forStatement()方法中添加代码：

```
      initializer = expressionStatement();
    }
    // 新增部分开始
    Expr condition = null;
    if (!check(SEMICOLON)) {
      condition = expression();
    }
    consume(SEMICOLON, "Expect ';' after loop condition.");
    // 新增部分结束
  }
```

> Again, we look for a semicolon to see if the clause has been omitted. The last clause is the increment.

同样，我们查找分号检查子句是否被忽略。最后一个子句是增量语句。

lox/Parser.java，在 forStatement()方法中添加：

```
    consume(SEMICOLON, "Expect ';' after loop condition.");
    // 新增部分开始
    Expr increment = null;
    if (!check(RIGHT_PAREN)) {
      increment = expression();
    }
    consume(RIGHT_PAREN, "Expect ')' after for clauses.");
    // 新增部分结束
  }
```

> It's similar to the condition clause except this one is terminated by the closing parenthesis. All that remains is the body.

它类似于条件式子句，只是这个子句是由右括号终止的。剩下的就是循环主体了。

lox/Parser.java，在 forStatement()方法中添加代码：

```
    consume(RIGHT_PAREN, "Expect ')' after for clauses.");
    // 新增部分开始
    Stmt body = statement();

    return body;
    // 新增部分结束
  }
```

> We’ve parsed all of the various pieces of the `for` loop and the resulting AST nodes are sitting in a handful of Java local variables. This is where the desugaring comes in. We take those and use them to synthesize syntax tree nodes that express the semantics of the `for` loop, like the hand-desugared example I showed you earlier.

我们已经解析了`for`循环的所有部分，得到的AST节点也存储在一些Java本地变量中。这里也是脱糖开始的地方。我们利用这些变量来合成表示`for`循环语义的语法树节点，就像前面展示的手工脱糖的例子一样。

> The code is a little simpler if we work backward, so we start with the increment clause.

如果我们从后向前处理，代码会更简单一些，所以我们从增量子句开始。

lox/Parser.java，在 forStatement()方法中新增：

```
    Stmt body = statement();
    // 新增部分开始
    if (increment != null) {
      body = new Stmt.Block(
          Arrays.asList(
              body,
              new Stmt.Expression(increment)));
    }
    // 新增部分结束
    return body;
```

> The increment, if there is one, executes after the body in each iteration of the loop. We do that by replacing the body with a little block that contains the original body followed by an expression statement that evaluates the increment.

如果存在增量子句的话，会在循环的每个迭代中在循环体结束之后执行。我们用一个代码块来代替循环体，这个代码块中包含原始的循环体，后面跟一个执行增量子语句的表达式语句。

lox/Parser.java，在 forStatement()方法中新增代码：

```
    }
    // 新增部分开始
    if (condition == null) condition = new Expr.Literal(true);
    body = new Stmt.While(condition, body);
    // 新增部分结束
    return body;
```

> Next, we take the condition and the body and build the loop using a primitive `while` loop. If the condition is omitted, we jam in `true` to make an infinite loop.

接下来，我们获取条件式和循环体，并通过基本的 `while` 语句构建对应的循环。如果条件式被省略了，我们就使用 `true` 来创建一个无限循环。

lox/Parser.java，在 forStatement()方法中新增：

```
    body = new Stmt.While(condition, body);
    // 新增部分开始
    if (initializer != null) {
      body = new Stmt.Block(Arrays.asList(initializer, body));
    }
    // 新增部分结束
    return body;
```

> Finally, if there is an initializer, it runs once before the entire loop. We do that by, again, replacing the whole statement with a block that runs the ini-

> tializer and then executes the loop.

最后，如果有初始化式，它会在整个循环之前运行一次。我们的做法是，再次用代码块来替换整个语句，该代码块中首先运行一个初始化式，然后执行循环。

> That’s it. Our interpreter now supports C-style `for` loops and we didn’t have to touch the Interpreter class at all. Since we desugared to nodes the interpreter already knows how to visit, there is no more work to do.

就是这样。我们的解释器现在已经支持了C语言风格的 `for` 循环，而且我们根本不需要修改解释器类。因为我们通过脱糖将其转换为了解释器已经知道如何访问的节点，所以无需做其它的工作。

> Finally, Lox is powerful enough to entertain us, at least for a few minutes. Here’s a tiny program to print the first 21 elements in the Fibonacci sequence:

最后，Lox已强大到足以娱乐我们，至少几分钟。下面是一个打印斐波那契数列前21个元素的小程序：

```
var a = 0;
var temp;

for (var b = 1; a < 10000; b = temp + b) {
  print a;
  temp = a;
  a = b;
}
```

#CHALLENGES

#习题

> 1、A few chapters from now, when Lox supports first-class functions and dynamic dispatch, we technically won't *need* branching statements built into the language. Show how conditional execution can be implemented in terms of those. Name a language that uses this technique for its control flow.

1、在接下来的几章中，当Lox支持一级函数和动态调度时，从技术上讲，我们就不需要在语言中内置分支语句。说明如何用这些特性来实现条件执行。说出一种在控制流中使用这种技术的语言。

> 2、Likewise, looping can be implemented using those same tools, provided our interpreter supports an important optimization. What is it, and why is it necessary? Name a language that uses this technique for iteration.

2、同样地，只要我们的解释器支持一个重要的优化，循环也可以用这些工具来实现。它是什么？为什么它是必要的？请说出一种使用这种技术进行迭代的语言。

> 3、Unlike Lox, most other C-style languages also support `break` and `continue` statements inside loops. Add support for `break` statements.
>
> The syntax is a `break` keyword followed by a semicolon. It should be a syntax error to have a `break` statement appear outside of any enclosing loop. At runtime, a `break` statement causes execution to jump to the end of the nearest enclosing loop and proceeds from there. Note that the `break` may be nested inside other blocks and `if` statements that also need to be exited.

3、与Lox不同，大多数其他C风格语言也支持循环内部的`break`和`continue`语句。添加对`break`语句的支持。

语法是一个`break`关键字，后面跟一个分号。如果`break`语句出现在任何封闭的循环之后，那就应该是一个语法错误。在运行时，`break`语句会跳转到最内层的封闭循环的末尾，并从那里开始继续执行。注意，`break`语句可以嵌套在其它需要退出的代码块和`if`语句中。

> ## #DESIGN NOTE: SPOONFULS OF SYNTACTIC SUGAR

#设计笔记：一些语法糖

> When you design your own language, you choose how much syntactic sugar to pour into the grammar. Do you make an unsweetened health food where each semantic operation maps to a single syntactic unit, or some decadent dessert where every bit of behavior can be expressed ten different ways? Successful languages inhabit all points along this continuum.
>
> On the extreme acrid end are those with ruthlessly minimal syntax like Lisp, Forth, and Smalltalk. Lispers famously claim their language “has no syntax”, while Smalltalkers proudly show that you can fit the entire grammar on an index card. This tribe has the philosophy that the *language* doesn’t need syntactic sugar. Instead, the minimal syntax and semantics it provides are powerful enough to let library code be as expressive as if it were part of the language itself.
>
> Near these are languages like C, Lua, and Go. They aim for simplicity and clarity over minimalism. Some, like Go, deliberately eschew both syntactic sugar and the kind of syntactic extensibility of the previous category. They want the syntax to get out of the way of the semantics, so they focus on keeping both the grammar and libraries simple. Code should be obvious more than beautiful.

> Somewhere in the middle you have languages like Java, C#, and Python. Eventually you reach Ruby, C++, Perl, and D—languages which have stuffed so much syntax into their grammar, they are running out of punctuation characters on the keyboard.
>
> To some degree, location on the spectrum correlates with age. It’s relatively easy to add bits of syntactic sugar in later releases. New syntax is a crowd pleaser, and it’s less likely to break existing programs than mucking with the semantics. Once added, you can never take it away, so languages tend to sweeten with time. One of the main benefits of creating a new language from scratch is it gives you an opportunity to scrape off those accumulated layers of frosting and start over.
>
> Syntactic sugar has a bad rap among the PL intelligentsia. There’s a real fetish for minimalism in that crowd. There is some justification for that. Poorly designed, unneeded syntax raises the cognitive load without adding enough expressiveness to carry its weight. Since there is always pressure to cram new features into the language, it takes discipline and a focus on simplicity to avoid bloat. Once you add some syntax, you’re stuck with it, so it’s smart to be parsimonious.
>
> At the same time, most successful languages do have fairly complex grammars, at least by the time they are widely used. Programmers spend a ton of time in their language of choice, and a few niceties here and there really can improve the comfort and efficiency of their work.
>
> Striking the right balance—choosing the right level of sweetness for your language—relies on your own sense of taste.

当你设计自己的语言时，你可以选择在语法中注入多少语法糖。你是要做一种不加糖、每个语法操作都对应单一的语法单元的健康食品？还是每一点行为都可以用10种不同方式实现的堕落的甜点？把这两种情况看作是两端的话，成功的语言分布在这个连续体的每个中间点。

极端尖刻的一侧是那些语法极少的语言，如Lisp、Forth和SmallTalk。Lisp的拥趸广泛声称他们的语言 "没有语法"，而Smalltalk的人则自豪地表示，你可以把

整个语法放在一张索引卡上。这个部落的理念是，语言不需要句法糖。相反，它所提供的最小的语法和语义足够强大，足以让库中的代码像语言本身的一部分一样具有表现力。

接近这些的是像C、Lua和Go这样的语言。他们的目标是简单和清晰，而不是极简主义。有些语言，如Go，故意避开了语法糖和前一类语言的语法扩展性。他们希望语法不受语义的影响，所以他们专注于保持语法和库的简单性。代码应该是明显的，而不是漂亮的。

介于之间的是Java、C#和Python等语言。最终，你会看到Ruby、C++、Perl和D-语言，它们在语法中塞入了太多的句法规则，以至于键盘上的标点符号都快用完了。

在某种程度上，频谱上的位置与年龄相关。在后续的版本中增加一些语法糖是比较容易的。新的语法很容易让人喜欢，而且与修改语义相比，它更不可能破坏现有的程序。一旦加进去，你就再也不能把它去掉了，所以随着时间的推移，语言会变得越来越甜。从头开始创建一门新语言的主要好处之一是，它给了你一个机会去刮掉那些累积的糖霜并重新开始。

语法糖在PL知识分子中名声不佳。那群人对极简主义有一种真正的迷恋。这是有一定道理的。设计不良的、不必要的语法增加了认知负荷，却没有增加相匹配的表达能力。因为一直会有向语言中添加新特性的压力，所以需要自律并专注于简单，以避免臃肿。一旦你添加了一些语法，你就会被它困住，所以明智的做法是要精简。

同时，大多数成功的语言都有相当复杂的语法，至少在它们被广泛使用的时候是这样。程序员在他们所选择的语言上花费了大量的时间，一些随处可见的细节确实可以提高他们工作时的舒适度和效率。

找到正确的平衡——为你的语言选择适当的甜度——取决于你自己的品味。

---

[^1]

其中最著名的就是罗素悖论。最初，集合理论允许你定义任何类型的集合。只要你能用英语描述它，它就是有效的。自然，鉴于数学家对自引用的偏爱，集合可以包含其他的集合。于是，罗素，这个无赖，提出了：

R是所有不包含自身的集合的集合。

R是否包含自己？如果不包含，那么根据定义的后半部分，它应该包含；如果包含，那么它就不满足定义。脑袋要炸了。

[^2]

图灵把他的发明称为 “a-machines”，表示“automatic(自动)”。他并没有自吹自擂到把自己的名字放入其中。后来的数学家们为他做了这些。这就是你如何在成名的同时还能保持谦虚。

[^3]

我们也基本上具备第三个条件了。你可以创建和拼接任意大小的字符串，因此也就可以存储无界内存。但我们还无法访问字符串的各个部分。

[^4]

条件操作符也称为三元操作符，因为它是C语言中唯一接受三个操作数的操作符。

[^5]

条件周围的圆括号只有一半是有用的。您需要在条件和then语句之间设置某种分隔符，否则解析器无法判断是否到达条件表达式的末尾。但是 `if` 后面的小括号并没有什么用处。Dennis Ritchie 把它放在那里是为了让他可以使用 `)` 作为结尾的分隔符，而且不会出现不对称的小括号。其他语言，比如Lua和一些BASICs，使用 `then` 这样的关键字作为结束分隔符，在条件表达式之前没有任何内容。而Go和Swift则要求语句必须是一个带括号的块，这样就可以使用语句开头的 `{` 来判断条件表达式是否结束。

[^6]

这个令人愉快的短语是由Peter J. Landin在1964年创造的，用来描述ALGOL等语言支持的一些很好的表达式形式是如何在更基本但可能不太令人满意的lambda演算的基础上增添一些小甜头的。