4 Scanning

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

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

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

4.1 The Interpreter Framework

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

This task has been variously called "scanning" and "lexing" (short for "lexical analysis") over the years. Way back when computers were as big as Winnebagos but had less memory than your watch, some people used "scanner" only to refer to the piece of code that dealt with reading raw source code characters from disk and buffering them in memory. Then "lexing" was the subsequent phase that did useful stuff with the characters.

These days, reading a source file into memory is trivial, so it's rarely a distinct phase in the compiler. Because of that, the two terms are basically interchangeable.

lox/Lox.java

```
package com.craftinginterpreters.lox;

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

public class Lox {
  public static void main(String[] args) throws IOException {
```

```
    if (args.length > 1) {
      System.out.println("Usage: jlox [script]");
      System.exit(64);
    } else if (args.length == 1) {
      runFile(args[0]);
    } else {
      runPrompt();
    }
  }
}
```

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

For exit codes, I'm using the conventions defined in the UNIX "sysexits.h" header. It's the closest thing to a standard I could find.

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

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

<< lox/Lox.java

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

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

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

<< lox/Lox.java

An interactive prompt is also called a "REPL" (pronounced like "rebel" but with a "p"). The name comes from Lisp where implementing one is as simple as wrapping a loop around a few built-in functions:

```
(print (eval (read)))
```

Working outwards from the most nested call, you **R**ead a line of input, **E**valuate it, **P**rint the result, then **L**oop and do it all over again.

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

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

function:

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

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

<< lox/Lox.java
add after runPrompt()

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

Error handling 4.1.1

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

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

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

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

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

<< lox/Lox.java
add after run()

Having said all that, for *this* interpreter, what we'll build is pretty bare bones. I'd love to talk about interactive debuggers, static analyzers, and other fun stuff, but there's only so much ink in the pen.

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

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

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

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

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

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

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

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

« lox/Lox.java in Lox

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

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

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

« lox/Lox.java in runFile

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

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

« lox/Lox.java

```
    }
```

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

Various phases of the front end will detect errors, but it's not really their job to know how to present that to a user. In a full-featured language implementation, you will likely have multiple ways errors get displayed: on stderr, in an IDE's error window, logged to a file, etc. You don't want that code smeared all over your scanner and parser.

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

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

I had exactly that when I first implemented jlox. I ended up tearing it out because it felt over-engineered for the minimal interpreter in this book.

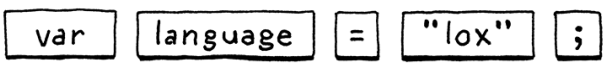
Lexemes and Tokens 4.2

Here's a line of Lox code:

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

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

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



The lexemes are only the raw substrings of the source code. However, in the process of grouping character sequences into lexemes, we also stumble upon

some other useful information. When we take the lexeme and bundle it together with that other data, the result is a token. It includes useful stuff like:

Token type 4.2.1

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

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

```
package com.craftinginterpreters.lox;

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

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

  // Literals.
  IDENTIFIER, STRING, NUMBER,

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

  EOF
}
```

<< After all, string comparison ends up looking at individual characters, and isn't that the scanner's job?

Literal value 4.2.2

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

object that will be used by the interpreter later.

Location information 4.2.3

Back when I was preaching the gospel about error handling, we saw that we need to tell users *where* errors occurred. Tracking that starts here. In our simple interpreter, we note only which line the token appears on, but more sophisticated implementations include the column and length too.

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

<< Token type

```
package com.craftinginterpreters.lox;

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

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

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

Some token implementations store the location as two numbers: the offset from the beginning of the source file to the beginning of the lexeme, and the length of the lexeme. The scanner needs to know these anyway, so there's no overhead to calculate them.

An offset can be converted to line and column positions later by looking back at the source file and counting the preceding newlines. That sounds slow, and it is. However, you need to do it *only when you need to actually display a line and column to the user*. Most tokens never appear in an error message. For those, the less time you spend calculating position information ahead of time, the better.

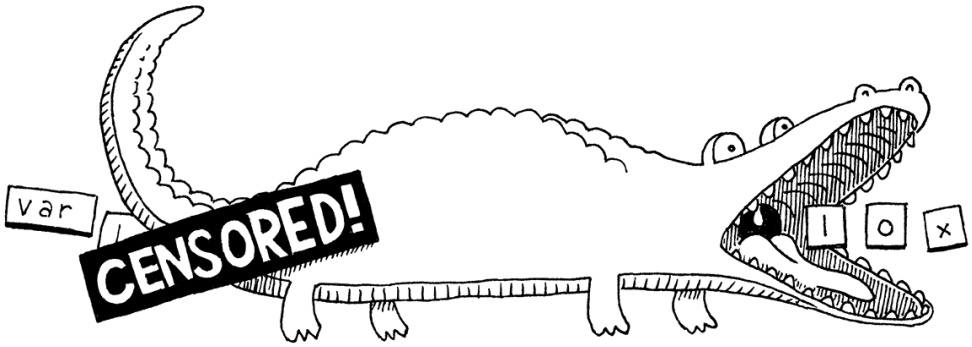
Now we have an object with enough structure to be useful for all of the later phases of the interpreter.

Regular Languages and Expressions 4.3

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

Then it loops back and does it again, starting from the very next character in the

source code. It keeps doing that, eating characters and occasionally, uh, excreting tokens, until it reaches the end of the input.



Lexical analygator.

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

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

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

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

It pains me to gloss over the theory so much, especially when it's as interesting as I think the Chomsky hierarchy and finite-state machines are. But the honest truth is other books cover this better than I could. *Compilers: Principles, Techniques, and Tools* (universally known as "the dragon book") is the canonical reference.

Lex was created by Mike Lesk and Eric Schmidt. Yes, the same Eric Schmidt who was executive chairman of Google. I'm not saying programming languages are a surefire path to wealth and fame, but we *can* count at least one mega billionaire among us.

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

The Scanner Class 4.4

Without further ado, let's make ourselves a scanner.

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

« lox/Scanner.java

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

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

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

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

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

I know static imports are considered bad style by some, but they save me from having to sprinkle `TokenType.` all over the scanner and parser. Forgive me, but every character counts in a book.

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

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

lox/Scanner.java
add after Scanner()

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

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

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

  Scanner(String source) {
```

lox/Scanner.java
in class Scanner

The `start` and `current` fields are offsets that index into the string. The `start` field points to the first character in the lexeme being scanned, and

`current` points at the character currently being considered. The `line` field tracks what source line `current` is on so we can produce tokens that know their location.

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

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

<< lox/Scanner.java
add after scanTokens()

Recognizing Lexemes 4.5

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

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

<< lox/Scanner.java
add after scanTokens()

Wondering why `/` isn't in here? Don't worry, we'll get to it.

Again, we need a couple of helper methods.

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

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

<< lox/Scanner.java
add after isAtEnd()

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

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

Lexical errors 4.5.1

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

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

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

<< lox/Scanner.java
in scanToken()

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

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

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

The code reports each invalid character separately, so this shotguns the user with a blast of errors if they accidentally paste a big blob of weird text. Coalescing a run of invalid characters into a single error would give a nicer user experience.

Operators 4.5.2

We have single-character lexemes working, but that doesn't cover all of Lox's operators. What about `!`? It's a single character, right? Sometimes, yes, but if the very next character is an equals sign, then we should instead create a `!=` lexeme.

Note that the `!` and `=` are *not* two independent operators. You can't write `! =` in Lox and have it behave like an inequality operator. That's why we need to scan `!=` as a single lexeme. Likewise, `<`, `>`, and `=` can all be followed by `=` to create the other equality and comparison operators.

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

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

      default:
```

<< lox/Scanner.java
in scanToken()

Those cases use this new method:

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

    current++;
    return true;
  }
```

<< lox/Scanner.java
add after scanToken()

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

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

Longer Lexemes 4.6

We're still missing one operator: `/` for division. That character needs a little

special handling because comments begin with a slash too.

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

      default:
```

<< lox/Scanner.java
in scanToken()

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

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

We've got another helper:

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

<< lox/Scanner.java
add after match()

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

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

Technically, `match()` is doing lookahead too. `advance()` and `peek()` are the fundamental operators and `match()` combines them.

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

```
        break;
```

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

      case '\n':
        line++;
        break;

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

<< lox/Scanner.java
in scanToken()

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

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

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

String literals 4.6.1

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

```
        break;

      case '"': string(); break;

      default:
```

<< lox/Scanner.java
in scanToken()

That calls:

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

<< lox/Scanner.java
add after scanToken()

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

    // The closing ".
    advance();

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

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

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

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

Number literals 4.6.2

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

```
1234
12.34
```

We don't allow a leading or trailing decimal point, so these are both invalid:

```
.1234
1234.
```

We could easily support the former, but I left it out to keep things simple. The

Since we look only for a digit to start a number, that means `-123` is not a number *literal*. Instead, `-123`, is an *expression* that applies `-` to the number literal `123`. In practice, the result is the same, though it has one interesting edge case if we were to add method calls on numbers. Consider:

```
print -123.abs();
```

This prints `-123` because negation has lower precedence than method calls. We could fix

latter gets weird if we ever want to allow methods on numbers like `123.sqrt()`.

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

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

This relies on this little utility:

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

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

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

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

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

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

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

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

that by making `-` part of the number literal. But then consider:

```
var n = 123;
print -n.abs();
```

This still produces `-123`, so now the language seems inconsistent. No matter what you do, some case ends up weird.

<< lox/Scanner.java
in scanToken()
replace 1 line

<< lox/Scanner.java
add after peek()

The Java standard library provides `Character.isDigit()`, which seems like a good fit. Alas, that method allows things like Devanagari digits, full-width numbers, and other funny stuff we don't want.

<< lox/Scanner.java
add after scanToken()

add:

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


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

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

I could have made `peek()` take a parameter for the number of characters ahead to look instead of defining two functions, but that would allow *arbitrarily* far lookahead. Providing these two functions makes it clearer to a reader of the code that our scanner looks ahead at most two characters.

Reserved Words and Identifiers 4.7

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

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

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

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

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

Consider this nasty bit of C code:

```
---a;
```

own use. That's where the term **reserved word** comes from.

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

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

The rest of the code lives over here:

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

  addToken(IDENTIFIER);
}
```

We define that in terms of these helpers:

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

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

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

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

static {
  keywords = new HashMap<>();
  keywords.put("and",    AND);
  keywords.put("class",  CLASS);
  keywords.put("else",   ELSE);
  keywords.put("false",  FALSE);
```

Is it valid? That depends on how the scanner splits the lexemes. What if the scanner sees it like this:

```
- --a;
```

Then it could be parsed. But that would require the scanner to know about the grammatical structure of the surrounding code, which entangles things more than we want. Instead, the maximal munch rule says that it is *always* scanned like:

```
-- -a;
```

It scans it that way even though doing so leads to a syntax error later in the parser.

```
    keywords.put("for",    FOR);
    keywords.put("fun",    FUN);
    keywords.put("if",     IF);
    keywords.put("nil",    NIL);
    keywords.put("or",     OR);
    keywords.put("print",  PRINT);
    keywords.put("return", RETURN);
    keywords.put("super",  SUPER);
    keywords.put("this",   THIS);
    keywords.put("true",   TRUE);
    keywords.put("var",    VAR);
    keywords.put("while",  WHILE);
  }
```

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

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

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

<< lox/Scanner.java in identifier()

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

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

CHALLENGES

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

2. Aside from separating tokens—distinguishing `print foo` from `printfoo`—spaces aren't used for much in most languages. However, in a couple of dark corners, a space *does* affect how code is parsed in CoffeeScript, Ruby, and the C preprocessor. Where and what effect does it have in each of those languages?

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

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

DESIGN NOTE: IMPLICIT SEMICOLONS

Programmers today are spoiled for choice in languages and have gotten picky about syntax. They want their language to look clean and modern. One bit of syntactic lichen that almost every new language scrapes off (and some ancient ones like BASIC never had) is `;` as an explicit statement terminator.

Instead, they treat a newline as a statement terminator where it makes sense to do so. The "where it makes sense" part is the challenging bit. While *most* statements are on their own line, sometimes you need to spread a single statement across a couple of lines. Those intermingled newlines should not be treated as terminators.

Most of the obvious cases where the newline should be ignored are easy to detect, but there are a handful of nasty ones:

- A return value on the next line:

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

  Is "value" the value being returned, or do we have a `return` statement with no value followed by an expression statement containing a string literal?

- A parenthesized expression on the next line:

```
func
(parenthesized)
```

  Is this a call to `func(parenthesized)`, or two expression statements, one for `func` and one for a parenthesized expression?

- A `-` on the next line:

```
first
-second
```

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

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

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

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

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

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

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

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

  An example in JavaScript:

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

  Here, the `console.log()` *expression* contains a function literal which in turn contains the *statement* `statement();`.

  Python would need a different set of rules for implicitly joining lines if you could get back *into* a statement where newlines should become meaningful while still

nested inside brackets.

- JavaScript's "automatic semicolon insertion" rule is the real odd one. Where other languages assume most newlines *are* meaningful and only a few should be ignored in multi-line statements, JS assumes the opposite. It treats all of your newlines as meaningless whitespace *unless* it encounters a parse error. If it does, it goes back and tries turning the previous newline into a semicolon to get something grammatically valid.

  This design note would turn into a design diatribe if I went into complete detail about how that even *works*, much less all the various ways that JavaScript's "solution" is a bad idea. It's a mess. JavaScript is the only language I know where many style guides demand explicit semicolons after every statement even though the language theoretically lets you elide them.

If you're designing a new language, you almost surely *should* avoid an explicit statement terminator. Programmers are creatures of fashion like other humans, and semicolons are as passé as ALL CAPS KEYWORDS. Just make sure you pick a set of rules that make sense for your language's particular grammar and idioms. And don't do what JavaScript did.

And now you know why Python's `lambda` allows only a single expression body.

NEXT CHAPTER: "REPRESENTING CODE" →