CHAPTER 2

Lexical Structure

The lexical structure of a programming language is the set of elementary rules that specifies how you write programs in that language. It is the lowest-level syntax of a language: it specifies what variable names look like, the delimiter characters for comments, and how one program statement is separated from the next, for example. This short chapter documents the lexical structure of JavaScript. It covers:

- Case sensitivity, spaces, and line breaks
- Comments
- Literals
- Identifiers and reserved words
- Unicode
- Optional semicolons

2.1 The Text of a JavaScript Program

JavaScript is a case-sensitive language. This means that language keywords, variables, function names, and other *identifiers* must always be typed with a consistent capitalization of letters. The `while` keyword, for example, must be typed "while," not "While" or "WHILE." Similarly, `online`, `Online`, `OnLine`, and `ONLINE` are four distinct variable names.

JavaScript ignores spaces that appear between tokens in programs. For the most part, JavaScript also ignores line breaks (but see §2.6 for an exception). Because you can use spaces and newlines freely in your programs, you can format and indent your programs in a neat and consistent way that makes the code easy to read and understand.

In addition to the regular space character (`\u0020`), JavaScript also recognizes tabs, assorted ASCII control characters, and various Unicode space characters as whitespace. JavaScript recognizes newlines, carriage returns, and a carriage return/line feed sequence as line terminators.

2.2 Comments

JavaScript supports two styles of comments. Any text between a `//` and the end of a line is treated as a comment and is ignored by JavaScript. Any text between the characters `/*` and `*/` is also treated as a comment; these comments may span multiple lines but may not be nested. The following lines of code are all legal JavaScript comments:

```
// This is a single-line comment.

/* This is also a comment */  // and here is another comment.

/*
 * This is a multi-line comment. The extra * characters at the start of
 * each line are not a required part of the syntax; they just look cool!
 */
```

2.3 Literals

A *literal* is a data value that appears directly in a program. The following are all literals:

```
12              // The number twelve
1.2             // The number one point two
"hello world"   // A string of text
'Hi'            // Another string
true            // A Boolean value
false           // The other Boolean value
null            // Absence of an object
```

Complete details on numeric and string literals appear in Chapter 3.

2.4 Identifiers and Reserved Words

An *identifier* is simply a name. In JavaScript, identifiers are used to name constants, variables, properties, functions, and classes and to provide labels for certain loops in JavaScript code. A JavaScript identifier must begin with a letter, an underscore (_), or a dollar sign ($). Subsequent characters can be letters, digits, underscores, or dollar signs. (Digits are not allowed as the first character so that JavaScript can easily distinguish identifiers from numbers.) These are all legal identifiers:

```
i
my_variable_name
v13
_dummy
$str
```

Like any language, JavaScript reserves certain identifiers for use by the language itself. These "reserved words" cannot be used as regular identifiers. They are listed in the next section.

2.4.1 Reserved Words

The following words are part of the JavaScript language. Many of these (such as `if`, `while`, and `for`) are reserved keywords that must not be used as the names of constants, variables, functions, or classes (though they can all be used as the names of properties within an object). Others (such as `from`, `of`, `get`, and `set`) are used in limited contexts with no syntactic ambiguity and are perfectly legal as identifiers. Still other keywords (such as `let`) can't be fully reserved in order to retain backward compatibility with older programs, and so there are complex rules that govern when they can be used as identifiers and when they cannot. (`let` can be used as a variable name if declared with `var` outside of a class, for example, but not if declared inside a class or with `const`.) The simplest course is to avoid using any of these words as identifiers, except for `from`, `set`, and `target`, which are safe to use and are already in common use.

```
as      const     export    get         null    target   void
async   continue  extends   if          of      this     while
await   debugger  false     import      return  throw    with
break   default   finally   in          set     true     yield
case    delete    for       instanceof  static  try
catch   do        from      let         super   typeof
class   else      function  new         switch  var
```

JavaScript also reserves or restricts the use of certain keywords that are not currently used by the language but that might be used in future versions:

```
enum  implements  interface  package  private  protected  public
```

For historical reasons, `arguments` and `eval` are not allowed as identifiers in certain circumstances and are best avoided entirely.

2.5 Unicode

JavaScript programs are written using the Unicode character set, and you can use any Unicode characters in strings and comments. For portability and ease of editing, it is common to use only ASCII letters and digits in identifiers. But this is a programming convention only, and the language allows Unicode letters, digits, and ideographs (but

not emojis) in identifiers. This means that programmers can use mathematical symbols and words from non-English languages as constants and variables:

```
const π = 3.14;
const sí = true;
```

2.5.1 Unicode Escape Sequences

Some computer hardware and software cannot display, input, or correctly process the full set of Unicode characters. To support programmers and systems using older technology, JavaScript defines escape sequences that allow us to write Unicode characters using only ASCII characters. These Unicode escapes begin with the characters `\u` and are either followed by exactly four hexadecimal digits (using uppercase or lowercase letters A–F) or by one to six hexadecimal digits enclosed within curly braces. These Unicode escapes may appear in JavaScript string literals, regular expression literals, and identifiers (but not in language keywords). The Unicode escape for the character "é," for example, is `\u00E9`; here are three different ways to write a variable name that includes this character:

```
let café = 1; // Define a variable using a Unicode character
caf\u00e9     // => 1; access the variable using an escape sequence
caf\u{E9}     // => 1; another form of the same escape sequence
```

Early versions of JavaScript only supported the four-digit escape sequence. The version with curly braces was introduced in ES6 to better support Unicode codepoints that require more than 16 bits, such as emoji:

```
console.log("\u{1F600}");  // Prints a smiley face emoji
```

Unicode escapes may also appear in comments, but since comments are ignored, they are simply treated as ASCII characters in that context and not interpreted as Unicode.

2.5.2 Unicode Normalization

If you use non-ASCII characters in your JavaScript programs, you must be aware that Unicode allows more than one way of encoding the same character. The string "é," for example, can be encoded as the single Unicode character `\u00E9` or as a regular ASCII "e" followed by the acute accent combining mark `\u0301`. These two encodings typically look exactly the same when displayed by a text editor, but they have different binary encodings, meaning that they are considered different by JavaScript, which can lead to very confusing programs:

```
const café = 1;  // This constant is named "caf\u{e9}"
const café = 2;  // This constant is different: "cafe\u{301}"
café  // => 1: this constant has one value
café  // => 2: this indistinguishable constant has a different value
```

The Unicode standard defines the preferred encoding for all characters and specifies a normalization procedure to convert text to a canonical form suitable for comparisons. JavaScript assumes that the source code it is interpreting has already been normalized and does *not* do any normalization on its own. If you plan to use Unicode characters in your JavaScript programs, you should ensure that your editor or some other tool performs Unicode normalization of your source code to prevent you from ending up with different but visually indistinguishable identifiers.

2.6 Optional Semicolons

Like many programming languages, JavaScript uses the semicolon (`;`) to separate statements (see Chapter 5) from one another. This is important for making the meaning of your code clear: without a separator, the end of one statement might appear to be the beginning of the next, or vice versa. In JavaScript, you can usually omit the semicolon between two statements if those statements are written on separate lines. (You can also omit a semicolon at the end of a program or if the next token in the program is a closing curly brace: }.) Many JavaScript programmers (and the code in this book) use semicolons to explicitly mark the ends of statements, even where they are not required. Another style is to omit semicolons whenever possible, using them only in the few situations that require them. Whichever style you choose, there are a few details you should understand about optional semicolons in JavaScript.

Consider the following code. Since the two statements appear on separate lines, the first semicolon could be omitted:

```
a = 3;
b = 4;
```

Written as follows, however, the first semicolon is required:

```
a = 3; b = 4;
```

Note that JavaScript does not treat every line break as a semicolon: it usually treats line breaks as semicolons only if it can't parse the code without adding an implicit semicolon. More formally (and with three exceptions described a bit later), JavaScript treats a line break as a semicolon if the next nonspace character cannot be interpreted as a continuation of the current statement. Consider the following code:

```
let a
a
=
3
console.log(a)
```

JavaScript interprets this code like this:

```
let a; a = 3; console.log(a);
```

JavaScript does treat the first line break as a semicolon because it cannot parse the code `let a a` without a semicolon. The second `a` could stand alone as the statement `a;`, but JavaScript does not treat the second line break as a semicolon because it can continue parsing the longer statement `a = 3;`.

These statement termination rules lead to some surprising cases. This code looks like two separate statements separated with a newline:

```
let y = x + f
(a+b).toString()
```

But the parentheses on the second line of code can be interpreted as a function invocation of `f` from the first line, and JavaScript interprets the code like this:

```
let y = x + f(a+b).toString();
```

More likely than not, this is not the interpretation intended by the author of the code. In order to work as two separate statements, an explicit semicolon is required in this case.

In general, if a statement begins with `(`, `[`, `/`, `+`, or `-`, there is a chance that it could be interpreted as a continuation of the statement before. Statements beginning with `/`, `+`, and `-` are quite rare in practice, but statements beginning with `(` and `[` are not uncommon at all, at least in some styles of JavaScript programming. Some programmers like to put a defensive semicolon at the beginning of any such statement so that it will continue to work correctly even if the statement before it is modified and a previously terminating semicolon removed:

```
let x = 0                           // Semicolon omitted here
;[x,x+1,x+2].forEach(console.log) // Defensive ; keeps this statement separate
```

There are three exceptions to the general rule that JavaScript interprets line breaks as semicolons when it cannot parse the second line as a continuation of the statement on the first line. The first exception involves the `return`, `throw`, `yield`, `break`, and `continue` statements (see Chapter 5). These statements often stand alone, but they are sometimes followed by an identifier or expression. If a line break appears after any of these words (before any other tokens), JavaScript will always interpret that line break as a semicolon. For example, if you write:

```
return
true;
```

JavaScript assumes you meant:

```
return; true;
```

However, you probably meant:

```
return true;
```

This means that you must not insert a line break between `return`, `break`, or `continue` and the expression that follows the keyword. If you do insert a line break, your code is likely to fail in a nonobvious way that is difficult to debug.

The second exception involves the ++ and -- operators (§4.8). These operators can be prefix operators that appear before an expression or postfix operators that appear after an expression. If you want to use either of these operators as postfix operators, they must appear on the same line as the expression they apply to. The third exception involves functions defined using concise "arrow" syntax: the => arrow itself must appear on the same line as the parameter list.

2.7 Summary

This chapter has shown how JavaScript programs are written at the lowest level. The next chapter takes us one step higher and introduces the primitive types and values (numbers, strings, and so on) that serve as the basic units of computation for JavaScript programs.