# 14 Scripting

## 14.3 Scripting the World Wide Web

Much of the content of the World Wide Web—particularly the content that is visible to search engines—is static: pages that seldom, if ever, change. But hypertext, the abstract notion on which the Web is based, was always conceived as a way to represent "the complex, the changing, and the indeterminate" [Nel65]. Much of the power of the Web today lies in its ability to deliver pages that move, play sounds, respond to user actions, or—perhaps most important—contain information created or formatted on demand, in response to the page-fetch request.

From a programming languages point of view, simple playback of recorded audio or video is not particularly interesting. We therefore focus our attention here on content that is generated on the fly by a program—a script—associated with an Internet URI (uniform resource identifier).[1] Suppose we type a URI into a browser on a client machine, and the browser sends a request to the appropriate web server. If the content is dynamically created, an obvious first question is: does the script that creates it run on the server or the client machine? These options are known as *server-side* and *client-side* web scripting, respectively.

Server-side scripts are typically used when the service provider wants to retain complete control over the content of the page, but can't (or doesn't want to) create the content in advance. Examples include the pages returned by search engines, Internet retailers, auction sites, and any organization that provides its clients with on-line access to personal accounts. Client-side scripts are typically used for tasks that don't need access to proprietary information, and are more efficient if executed on the client's machine. Examples include interactive animation, error-checking of fill-in forms, and a wide variety of other self-contained calculations.

---

1 The term "URI" is often used interchangeably with "URL" (uniform resource locator), but the World Wide Web Consortium distinguishes between the two. All URIs are hierarchical (multipart) names. URLs are one kind of URIs; they use a naming scheme that indicates where to find the resource. Other URIs can use other naming schemes.

```
#!/usr/bin/perl

print "Content-type: text/html\n\n";
print "<!DOCTYPE html>\n";

print "<html lang=\"en\">\n";
$host = `hostname`; chop $host;
print "<head>\n";
print "<meta charset=\"utf-8\">\n";
print "<title>Status of ", $host, "</title>\n";
print "</head>\n<body>\n";
print "<h1>", $host, "</h1>\n";
print "<pre>\n", `uptime`, "\n", `who`;
print "</pre>\n</body>\n</html>\n";
```

**Figure 14.14** A simple CGI script in Perl. If this script is named `status.perl`, and is installed in the server's `cgi-bin` directory, then a user anywhere on the Internet can obtain summary statistics and a list of users currently logged into the server by typing *hostname/cgi-bin/status.perl* into a browser window.

### 14.3.1 CGI Scripts

The original mechanism for server-side web scripting was the Common Gateway Interface (CGI). A CGI script is an executable program residing in a special directory known to the web server program. When a client requests the URI corresponding to such a program, the server executes the program and sends its output back to the client. Naturally, this output needs to be something that the browser will understand—typically HTML.

CGI scripts may be written in any language available on the server's machine, though Perl is particularly popular: its string-handling and "glue" mechanisms are ideally suited to generating HTML, and it was already widely available during the early years of the Web. As a simple if somewhat artificial example, suppose we would like to be able to monitor the status of a server machine shared by some community of users. The Perl script in Figure C-14.14 creates a web page titled by the name of the server machine, and containing the output of the `uptime` and `who` commands (two simple sources of status information). The script's initial `print` command produces an HTTP message header, indicating that what follows is HTML. Sample output from executing the script appears in Figure C-14.15. ■

**EXAMPLE 14.77**
Remote monitoring with a CGI script

**EXAMPLE 14.78**
Adder web form with a CGI script

CGI scripts are commonly used to process on-line forms. A simple example appears in Figure C-14.16. The `form` element in the HTML file specifies the URI of the CGI script, which is invoked when the user hits the Submit button. Values previously entered into the `input` fields are passed to the script either as a trailing part of the URI (for a `get`-type form) or on the standard input stream (for a `post`-type form, shown here).[2] With either method, we can access the values using the

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Status of sigma.cs.rochester.edu</title>
</head>
<body>
<h1>sigma.cs.rochester.edu</h1>
<pre>
22:10  up 5 days, 12:50, 5 users, load averages: 0.40 0.37 0.31

scott    console  Feb 13 09:21
scott    ttyp2    Feb 17 15:27
test     ttyp3    Feb 18 17:10
test     ttyp4    Feb 18 17:11
</pre>
</body>
</html>
```

Status of sigma.cs.rochester.edu

## sigma.cs.rochester.edu

```
22:10  up 5 days, 12:50, 5 users, load averages:  0.40 0.37 0.31

scott    console  Feb 13 09:21
scott    ttyp2    Feb 17 15:27
test     ttyp3    Feb 18 17:10
test     ttyp4    Feb 18 17:11
```

**Figure 14.15** **Sample output from the script of Figure C-14.14.** HTML source appears at top; the rendered page is below.

`param` routine of the standard `CGI` Perl library, loaded at the beginning of our script.

### 14.3.2 Embedded Server-Side Scripts

Though widely used, CGI scripts have several disadvantages:

- The web server must launch each script as a separate program, with potentially significant overhead (though a CGI script compiled to native code can be very fast once running).

**2** One typically uses `post` type forms for one-time requests. A `get` type form appears a little clumsier, because arguments are visibly embedded in the URI, but this gives it the advantage of repeatability: it can be "bookmarked" by client browsers.

```
<!DOCTYPE html>
<html lang="en">
<head><meta charset="utf-8"><title>Adder</title></head>
<body>
<form action="/cgi-bin/add.perl" method="post">
<p><input name="argA" size=3>First addend<br>
   <input name="argB" size=3>Second addend</p>
<p><input type="submit"></p>
</form>
</body>
</html>
```

Adder

12 First addend
34 Second addend

Submit

---

```
#!/usr/bin/perl

use CGI qw(:standard);      # provides access to CGI input fields
$argA = param("argA");  $argB = param("argB");  $sum = $argA + $argB;

print "Content-type: text/html\n\n";
print "<!DOCTYPE html>\n";

print "<html lang=\"en\">\n";
print "<head><meta charset=\"utf-8\"><title>Sum</title></head>\n<body>\n";
print "<p>$argA plus $argB is $sum</p>\n";
print "</body>\n</html>\n";
```

---

```
<!DOCTYPE html>
<html lang="en">
<head><meta charset="utf-8"><title>Sum</title></head>
<body>
<p>12 plus 34 is 46</p>
</body>
</html>
```

Sum

12 plus 34 is 46

**Figure 14.16** An interactive CGI form. Source for the original web page is shown at the upper left, with the rendered page to the right. The user has entered 12 and 34 in the text fields. When the Submit button is pressed, the client browser sends a request to the server for URI */cgi-bin/add.perl*. The values 12 and 13 are contained within the request. The Perl script, shown in the middle, uses these values to generate a new web page, shown in HTML at the bottom left, with the rendered page to the right.

- Because the server has little control over the behavior of a script, scripts must generally be installed in a trusted directory by trusted system administrators; they cannot reside in arbitrary locations as ordinary pages do.
- The name of the script appears in the URI, typically prefixed with the name of the trusted directory, so static and dynamic pages look different to end users.

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Status of <?php echo $host = chop(`hostname`) ?></title>
</head>
<body>
<h1><?php echo $host ?></h1>
<pre>
<?php echo `uptime`, "\n", `who` ?>
</pre>
</body>
</html>
```

**Figure 14.17** **A simple PHP script embedded in a web page.** When served by a PHP-enabled host, this page performs the equivalent of the CGI script of Figure C-14.14.

- Each script must generate not only dynamic content but also the HTML tags that are needed to format and display it. This extra "boilerplate" makes scripts more difficult to write.

To address these disadvantages, most web servers provide a "module-loading" mechanism that allows interpreters for one or more scripting languages to be incorporated into the server itself. Scripts in the supported language(s) can then be embedded in "ordinary" web pages. The web server interprets such scripts directly, without launching an external program. It then replaces the scripts with the output they produce, before sending the page to the client. Clients have no way to even know that the scripts exist.

Embeddable server-side scripting languages include PHP, PowerShell (in Microsoft Active Server Pages), Ruby, Cold Fusion (from Macromedia Corp.), and Java (via "Servlets" in Java Server Pages). The most common of these is PHP. Though descended from Perl, PHP has been extensively customized for its target domain, with built-in support for (among other things) email and MIME encoding, all the standard Internet communication protocols, authentication and security, HTML and URI manipulation, and interaction with dozens of database systems.

**EXAMPLE 14.79**
Remote monitoring with a PHP script

The PHP equivalent of Figure C-14.14 appears in Figure C-14.17. Most of the text in this figure is standard HTML. PHP code is embedded between `<?php` and `?>` delimiters. These delimiters are not themselves HTML; rather, they indicate a *processing instruction* that needs to be executed by the PHP interpreter to generate replacement text. The "boilerplate" parts of the page can thus appear verbatim; they need not be generated by `print` (Perl) or `echo` (PHP) commands. Note that the separate script fragments are part of a single program. The `$host` variable, for example, is set in the first fragment and used again in the second.

**EXAMPLE 14.80**
A fragmented PHP script

PHP scripts can even be broken into fragments in the middle of structured statements. Figure C-14.18 contains a script in which `if` and `for` statements span fragments. In effect, the HTML text between the end of one script fragment and the beginning of the next behaves as if it had been output by an `echo` command.

```
<!DOCTYPE html>
<html lang="en">
<head><meta charset="utf-8"><title>20 numbers</title></head>
<body>
<p>
<?php
         for ($i = 0; $i < 20; $i++) {
             if ($i % 2) { ?>
<b><?php
                 echo " $i"; ?>
</b><?php
             } else echo " $i";
         }
 ?>
</p>
</body>
</html>
```

**Figure 14.18** **A fragmented PHP script.** The `if` and `for` statements work as one might expect, despite the intervening raw HTML. When requested by a browser, this page displays the numbers from 0 to 19, with odd numbers written in bold.

Web designers are free to use whichever approach (`echo` or escape to raw HTML) seems most convenient for the task at hand. ■

#### *Self-Posting Forms*

**EXAMPLE 14.81**
Adder web form with a PHP script

By changing the `action` attribute of the `FORM` element, we can arrange for the Adder page of Figure C-14.16 to invoke a PHP script instead of a CGI script:

```
<form action="add.php" method="post">
```

The PHP script itself is shown in the top half of Figure C-14.19. Form values are made available to the script in an associative array (hash table) named `_REQUEST`. No special library is required. ■

**EXAMPLE 14.82**
Self-posting Adder web form

Because our PHP script is executed directly by the web server, it can safely reside in an arbitrary web directory, including the one in which the Adder page resides. In fact, by checking to see how a page was requested, we can merge the form and the script into a single page, and let it service its own requests! We illustrate this option in the bottom half of Figure C-14.19. ■

### 14.3.3 Client-Side Scripts

While embedded server-side scripts are generally faster than CGI scripts, at least when start-up cost predominates, communication across the Internet is still too slow for truly interactive pages. If we want the behavior or appearance of the page

```
<!DOCTYPE html>
<html lang="en">
<head><meta charset="utf-8"><title>Adder</title></head>
<body><p>
<?php
      $argA = $_REQUEST['argA']; $argB = $_REQUEST['argB'];
      $sum = $argA + $argB;
      echo "$argA plus $argB is $sum\n";
 ?>
</p></body></html>
```

---

```
<!DOCTYPE html>
<html lang="en">
<head><meta charset="utf-8">
<?php
    $argA = $_REQUEST['argA'];  $argB = $_REQUEST['argB'];
    if ($argA == "" || $argB == "") {
?>
        <title>Adder</title></head><body>
        <form action="adder.php" method="post">
        <p><input name="argA" size="3"> First addend<br>
           <input name="argB" size="3"> Second addend</p>
        <p><input type="submit"></p>
        </form></body></html>
<?php
    } else {
?>
        <title>Sum</title></head><body><p>
<?php
        $sum = $argA + $argB;
        echo "$argA plus $argB is $sum\n";
?>
        </p></body></html>
<?php
    }
?>
```

**Figure 14.19** **An interactive PHP web page.** The script at top could be used in place of the script in the middle of Figure C-14.16. The lower script in the current figure replaces both the web page at the top and the script in the middle of Figure C-14.16. It checks to see if it has received a full set of arguments. If it hasn't, it displays the fill-in form; if it has, it displays results.

to change as the user moves the mouse, clicks, types, or hides or exposes windows, we really need to execute some sort of script on the client's machine.

Because they run on the web designer's site, CGI scripts and, to a lesser extent, embeddable server-side scripts can be written in many different languages. All the client ever sees is standard HTML. Client-side scripts, by contrast, require an interpreter on the client's machine. By virtue of having been "in the right place at the right time" historically, JavaScript is supported with at least some degree of consistency by almost all of the world's web browsers. Moreover, given the number of legacy browsers still running, and the difficulty of convincing users to upgrade or to install new plug-ins, it has been difficult for any other option for client-side scripting to gain traction. Only recently, with the advent of WebAssembly, has the dominance of JavaAcript begun to wane.

**EXAMPLE 14.83**
Adder web form in JavaScript

Figure C-14.20 shows a page with embedded JavaScript that imitates (on the client) the behavior of the Adder scripts of Figures C-14.16 and C-14.19. Function `doAdd` is defined in the header of the page so it is available throughout. In particular, it will be invoked when the user clicks on the Calculate button. By default, the input values are character strings; we use the `parseInt` function to convert them to integers. The parentheses around `(argA + argB)` in the final assignment statement then force the use of integer addition. The other occurrences of + are string concatenation. To disable the usual mechanism whereby input data are submitted to the server when the user hits the enter or return key, we have specified a dummy behavior for the `onsubmit` attribute of the form.

Rather than replace the page with output text, as our CGI and PHP scripts did, we have chosen in our JavaScript version to append the output at the bottom. The HTML `SPAN` element provides a named place in the document where this output can be inserted, and the `getElementById` JavaScript method provides us with a reference to this element. The HTML *Document Object Model (DOM)*, standardized by the World Wide Web Consortium (W3C), specifies a very large number of other elements, attributes, and user actions, all of which are accessible in JavaScript. Through them scripts can, at appropriate times, inspect or alter almost any aspect of the content, structure, or style of a page.

### 14.3.4 Java Applets and Other Embedded Elements

As an alternative to requiring client-side scripts to interact with the DOM of a web page, many browsers once supported an *embedding* mechanism that allowed a browser plug-in to assume responsibility for some rectangular region of the page, in which it could then display whatever it wanted. In other words, plug-ins were less a matter of scripting the browser than of bypassing it entirely. Historically, they were widely used for content—animations and video in particular—that were poorly supported by early versions of HTML.

**EXAMPLE 14.84**
Embedding an applet in a web page

Programs designed to be run by a Java plug-in were commonly known as *applets*. Consider, for example, an applet to display a clock with moving hands. Legacy browsers supported several different applet tags, but as of HTML5 the standard syntax looked like this:

```
<!DOCTYPE html>
<html lang="en">
<head><meta charset="utf-8"><title>Adder</title>
<script type="text/javascript">
function doAdd() {
    argA = parseInt(document.adder.argA.value)
    argB = parseInt(document.adder.argB.value)
    x = document.getElementById('sum')
    while (x.hasChildNodes())
        x.removeChild(x.lastChild)  // delete old content
    t = document.createTextNode(argA + " plus "
        + argB + " is " + (argA + argB))
    x.appendChild(t)
}
</script>
</head>
<body>
<form name="adder" onsubmit="return false">
<p><INPUT name="argA" size=3> First addend<br>
   <INPUT name="argB" size=3> Second addend</p>
<p><input type="button" onclick="doAdd()" value="Calculate"></p>
</form>
<p><span id="sum"></span></p>
</body>
</html>
```

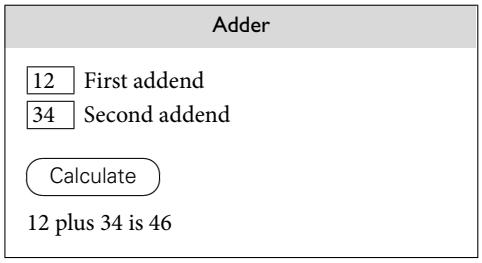



**Figure 14.20** **An interactive JavaScript web page.** Source appears at left. The rendered version on the right shows the appearance of the page after the user has entered two values and hit the Calculate button, causing the output message to appear. By entering new values and clicking again, the user can calculate as many sums as desired. Each new calculation will replace the output message.

```
<embed type="application/x-java-applet" code="Clock.class">
```

The `type` attribute informed the browser that the embedded element was expected to be a Java applet; the `code` element provided the applet's URI. Additional attributes could be used to specify such properties as the required interpreter version number and the size of the needed display space.

As one might infer from the existence of the `type` attribute, `embed` tags (and similar `object` tags) can request execution by a variety of plug-ins—not just a Java Virtual Machine. Historically, the most widely used plug-in was Adobe's Flash Player. Though scriptable, Flash Player is more accurately described as a multimedia display engine than a general purpose programming language interpreter.

Over time, plug-ins have proven to be a major source of browser security bugs. Almost any nontrivial plug-in requires access to operating system services—network IO, local file space, graphics acceleration, and so on. Providing just enough service to make the plug-in useful—but not enough to allow it to do any harm—has proven extremely difficult. To address this problem, extensive multimedia support has been built into HTML5, allowing the browser itself to assume responsibility for

much of what was once accomplished with plug-ins. Security is still a problem, but the number of software modules that must be trusted—and the number of points at which an attacker might try to gain entrance—is significantly reduced. Almost all browsers now disable Java by default. Most disable Flash as well.

### ✓ CHECK YOUR UNDERSTANDING

47. Explain the distinction between *server-side* and *client-side* web scripting.
48. List the tradeoffs between CGI scripts and embedded PHP.
49. Why are CGI scripts usually installed only in a special directory?
50. Explain how a PHP page can service its own requests.
51. Why might we prefer to execute a web script on the server rather than the client? Why might we sometimes prefer the client instead?
52. What is the HTML *Document Object Model*? What is its significance for client-side scripting?
53. What is the relationship between JavaScript and Java?
54. What is an *applet*? Why are applets usually not considered an example of scripting?
55. Why are Java applets and Flash objects no longer commonly supported by web browsers?

### DESIGN & IMPLEMENTATION

#### 14.12 JavaScript and Java

Despite its name, JavaScript has no connection to Java beyond some superficial syntactic similarity. The language was originally developed by Brendan Eich at Netscape Corp. in 1995. Eich called his creation *LiveScript*, but the company chose to rename it as part of a joint marketing agreement with Sun Microsystems, prior to its public release. Trademark on the JavaScript name is actually owned by Oracle, which acquired Sun in 2010.

Netscape's browser was the market leader in 1995, and JavaScript usage grew extremely fast. To remain competitive, developers at Microsoft added JavaScript support to Internet Explorer, but they used the name *JScript* instead, and they introduced a number of incompatibilities with the Netscape version of the language. A common version was standardized as *ECMAScript* by the European standards body in 1997 (and subsequently by the ISO), but major incompatibilities remained in the Document Object Models provided by different browsers. These have been gradually resolved through a series of standards from the W3C and WHATWG, but legacy pages and legacy browsers continue to plague web developers.

### 14.3.5 XSLT

Most readers will undoubtedly have had the opportunity to write—or at least to read—the HTML (hypertext markup language) used to compose web pages. HTML has, for the most part, a nested structure in which fragments of documents (*elements*) are delimited by *tags* that indicate their purpose or appearance. We saw in Section 14.2.2, for example, that top-level headings are delimited with `<h1>` and `</h1>`. HTML was inspired by an older standard known as SGML (standard generalized markup language), developed in the 1980s and used, among other things, to computerize both the Oxford English Dictionary and the technical documentation of Boeing Corp.

**DESIGN & IMPLEMENTATION**

#### 14.13 How far can you trust a script?

Security becomes an issue whenever code is executed using someone else's resources. On a hosting machine, web servers are usually installed with very limited access rights, and with only a limited view of the host's file system. This strategy limits the set of pages accessible through the server to a well-defined subset of what would be visible to users logged into the hosting machine directly. By contrast, CGI scripts are separate executable programs, and can potentially run with the privileges of whoever installs them. To prevent users on the hosting machine from accidentally or intentionally passing their privileges to arbitrary users on the Internet, most system administrators configure their machines so that CGI scripts must reside in a special directory, and be installed by a trusted user. Embedded server-side scripts can reside in any file because they are guaranteed to run with the (limited) rights of the server itself.

A larger risk is posed by code downloaded over the Internet and executed on a client machine. Because such code is in general untrusted, it must be executed in a carefully controlled environment, sometimes called a *sandbox* (a place where a child can safely play), to prevent it from doing any damage. As a general rule, embedded JavaScript cannot access the local file system, memory management system, or network, nor can it manipulate documents from other sites. Java applets, likewise, have only limited ability to access external resources. Reality is a bit more complicated, of course: Sometimes a script needs access to, say, a temporary file of limited size, or a network connection to a trusted server. Mechanisms exist to certify sites as *trusted*, or to allow a trusted site to certify the trustworthiness of pages from other sites. Scripts on pages obtained through a trusted mechanism may then be given extended rights. Such mechanisms must be used with care. Finding the right balance between security and functionality remains one of the central challenges of the Web, and of distributed computing in general. (More on this topic can be found in Sections 15.2.3 and 16.2.4, and in Explorations 16.21 and 16.22.)

In the early days of the Web, SGML was clearly too complex and formal for web pages, which needed to be written by hand and rendered in real time by slow computers. The simpler HTML evolved in an informal and ad hoc way, with incompatible extensions made by competing vendors. Standardization has been a long and difficult process: incompatibilities among browsers continue to frustrate web designers, and several features of the language that have been deprecated[3] in the most recent standards are nonetheless still widely used. Other features, while not deprecated, are widely regarded in hindsight to have been mistakes.

**EXAMPLE 14.85**
Content versus presentation in HTML

Probably the biggest problem with HTML is that it does not adequately distinguish between the *content* and the *presentation* (appearance) of a document. As a trivial example, web designers sometimes use `<i>...</i>` tags to request that text be set in an italic font, when `<em>...</em>` (emphasis) would be more appropriate. A browser for the visually impaired might choose to emphasize text with something other than italics, and might render book titles (also often specified with `<i>...</i>`) in some entirely different fashion. More significantly, many web designers use tables (`<table>...</table>`) to control the relative positioning of elements on a page, when the content isn't tabular at all. As the Web has extended across cell phones, televisions, tablets, watches, and audio-only devices, the need to distinguish between content and presentation has become increasingly essential.

This is where XML stepped in. A streamlined descendant of SGML, developed by the W3C in the mid to late 1990s, XML has at least three important advantages over HTML for data and document representation: (1) its syntax and semantics are more regular and consistent, and more consistently implemented across platforms; (2) it is *extensible*, meaning that users can define new tags; (3) it specifies content only, leaving presentation to a companion standard known as XSL (extensible stylesheet language). XSLT is a portion of XSL devoted to *transforming* XML: selecting, reorganizing, and modifying tags and the elements they delimit—in effect, scripting the processing of data represented in XML.

### *Internet Alphabet Soup*

Learning about web standards can be a daunting task: there is an enormous number of buzzwords, standards, and multiletter abbreviations. The standards—and the relationships among them—are also moving targets, promulgated by groups whose interests are not always in sync. To start, it may help to note that each of the major markup languages—SGML, HTML, and XML—has a corresponding *stylesheet language*: DSSSL, CSS, and XSL, respectively. A stylesheet language is used to control the presentation of a document, separate from its content. Stylesheet languages are essential for SGML and XML; without them there is no way to know whether a `<RECORD>` represents a database entry, an antique phonograph album, or an Olympic achievement, much less how to display it. HTML is less dependent

**3** A *deprecated* feature is one whose use is officially discouraged, but permitted on a temporary basis, to ease the transition to new and presumably better alternatives.

on stylesheets, but most professionally maintained web sites use CSS to create a uniform "look and feel" across a collection of pages without embedding redundant information in every page.

SGML is still used for large-scale projects in the business world, though many newer projects have chosen to use XML or JSON, the JavaScript Object Notation. JSON is more compact and self-descriptive than XML, and is commonly used to transmit structured data between web servers and clients. It does not have a stylesheet language comparable to XSL, however. HTML continues to evolve (see sidebar C-14.14). HTML5, codified by the World Wide Web Consortium in 2014, added extensive support for multimedia content, and specified both general and XML-compliant versions of the syntax.

### *XML and XHTML*

As a general rule, the syntax of XML is simpler than that of SGML or HTML. To allow XML tools (XSLT in particular) to be used to process web pages, the HTML5 standard defines a restricted version of the HTML syntax, known as XHTML. With a few minor exceptions, any web page that can be specified in HTML can also be specified in XHTML, and vice versa. The `content-type` header that precedes a web page when transmitted over the Internet tells the browser which parser to use:

**DESIGN & IMPLEMENTATION**

#### 14.14 W3C and WHATWG

Standardization efforts for HTML have a complicated history. With the completion in 1998 of the XML 1.0 specification, the World Wide Web Consortium (W3C) focused on XHTML, in an effort to push the world toward a "cleaned-up," XML-compliant version of HTML. Over the next few years, this strategy proved increasingly contentious. In 2004, a group of influential individuals from Apple, Mozilla, and Opera split off to form a separate Web Hypertext Application Technology Working Group (WHATWG), with the goal of evolving HTML in a way that preserved complete backward compatibility and interoperability. In 2006, the W3C reconsidered its position, and began to work with WHATWG toward what eventually became HTML5. Both groups continued to work on HTML evolution, largely but not entirely in sync. In 2019, they agreed that future development would belong to WHATWG, though W3C continues to participate.

While W3C would prefer a dated, finalized document (which it would number HTML5), WHATWG's "Living Standard" for HTML has been continuously evolving (without version numbers) since 2012. WHATWG believes that the standard should reflect without necessarily dictating current practice, as embodied in the browsers of all major vendors. Both the W3C and WHATWG distinguish carefully between what a conforming document should contain and what a conforming browser should be able to render: the latter is significant superset of the former.

`text/html` means "regular" HTML; `application/xhtml+xml` means XHTML. In practice, the principal differences between the notations are that XHTML is harder for human beings to write, because the rules are stricter, and XML parsers are designed to reject (and decline to render) any page that is not *well formed* (syntactically correct). HTML parsers are designed to tolerate—and do something reasonable with—even the worst "tag soup." With some care, it is possible to write pages that will be processed correctly by both HTML and XHTML parsers; such pages are said to use *polyglot markup* (syntax).

In any well-formed XML document (including those written in XHTML), tags must either constitute properly nested, matched pairs, or be explicit singletons, which end with a "`/>`" delimiter. Similarly, the values of *attributes* (key–value pairs embedded within tags) must always be specified with quotes. The following fragment, for example, is well formed (though incomplete) XHTML:

**EXAMPLE 14.86**
Well-formed XHTML

```
<em><q id="favorite">I defy the tyranny of precedent</q></em><br />
(Clara Barton)
```

Here the quotation element (`<q>` … `</q>`) is nested inside the emphasis element (`<em>` … `</em>`). Moreover the "break" element (`<br />`), which usually causes subsequent text to start on a new line, is explicitly a singleton; it has a slash before its closing "`>`" delimiter. (To avoid confusing certain legacy browsers, one sometimes needs a space in front of the slash.) The example fragment would be malformed if the slash were missing, if the opening `<em><q>` tags were reversed (`<q><em>`), or if the attribute value `"favorite"` had not be enclosed in quote marks. An HTML parser would tolerate these errors; an XML parser will not. ■

The set of tags to be used in an XML document can be specified by naming a *document type definition* (DTD) in the document's `DOCTYPE` header, or by naming an *XML Schema* in an attribute of the document's top-level tag. (XML Schemas are a newer format, but DTDs remain in widespread use.) Among other things, a DTD or Schema indicates which tags are allowed, whether those tags are pairs or singletons, whether they permit attributes, and whether any attributes are mandatory. If a document has no DTD or Schema, it is said to define a DTD *implicitly* by virtue of which tags are actually used. Implicit definition suffices for the examples in this chapter.

Because tags must nest in XML, a document has a natural tree-based structure. Figure C-14.21 shows the source for a small but complete polyglot HTML5 document, together with the tree it represents. There are three kinds of nodes in the tree: elements (delimited by tags in the source), text, and attributes. The internal (nonleaf) nodes are all elements. Everything nested between the beginning and ending tags of an element is an attribute or child of that element in the tree.

**EXAMPLE 14.87**
XHTML to display a favorite quote

The root of our document, named "`/`" by convention, has one child—the `html` element. This in turn has three attributes—`xmlns`, `lang`, and `xml:lang`—and two child elements—`head` and `body`. The `xmlns` attribute specifies a URI for our document's *namespace*. This serves a purpose similar to that of C++ namespaces or Java packages (Section 3.8): it allows us to give tag names a disambiguating

```
<!DOCTYPE html>
<html xmlns="https://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
<head>
<meta charset="UTF-8" />
<title>Favorite Quote</title>
</head>
<body>
<p>
<em><q id="favorite">I defy
the tyranny of precedent</q></em><br />
(Clara Barton)
</p>
</body>
</html>
```

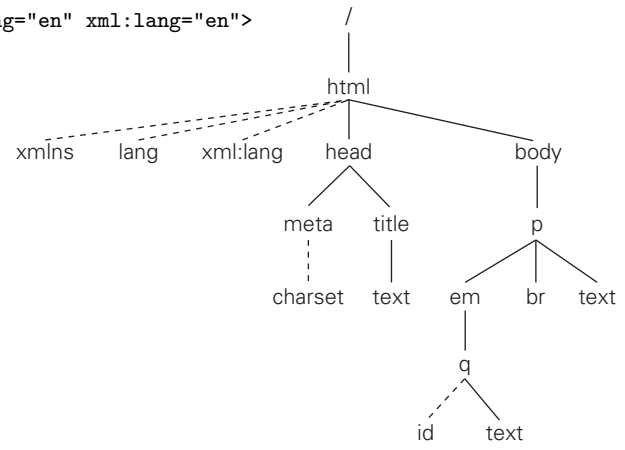



**Figure 14.21** **A complete XHTML document and its corresponding tree.** Child elements are shown with solid lines, attributes with dashed lines.

prefix: `xhtml:table` versus `furniture:table`. With the value we have specified for the `xmlns` attribute, any tag in the document that doesn't have a prefix will automatically be interpreted as being in the `xhtml` namespace. The `lang` and `xml:lang` tags specify the source language (English) for HTML and XML parsers, respectively.

### *XSLT and XPath*

XSL (extensible stylesheet language) can be thought of as a language for specifying what to *do* with an XML document. It has four sublanguages, called XSLT, XPath, XSL-FO, and XQuery. XSLT is a scripting language that takes XML as input and produces textual output—often transformed XML or HTML, but potentially other formats as well.

**EXAMPLE 14.88**
XPath names for XHTML elements

XPath is a language used to name things in XML documents. XPath names frequently appear in the attributes of XSLT elements. Returning to Figure C-14.21, the quotation element of our document could be named in XPath as `/html/body/p/em/q`. The emphasis element and its break and text-node siblings, together, could be named as `/html/body/p/*`. XPath includes a rich set of naming mechanisms, including absolute (from the root) and relative (from the current node) navigation, wildcards, predicates, substring and regular expression manipulation, and counting and arithmetic functions. We will see some of these in the extended example below.

XSL-FO (XSL formatting objects) is a set of tags to specify the *layout* (presentation) of a document, in terms of pages, regions (e.g., header, body, footer), blocks (paragraph, table, list), lines, and in-line elements (character, image). An XSLT script might be used to add XSL-FO tags to an XML document, or to transform a document that already has XSL-FO tags in it—perhaps to split a long single-page

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="bib.xsl"?>
<bibliography>
  <book>
    <author>Guido van Rossum</author>
    <editor>Fred L. Drake, Jr.</editor>
    <title>The Python Language Reference Manual (version 3.2)</title>
    <publisher>Network Theory, Ltd.</publisher>
    <address>Bristol, UK</address>
    <year>2011</year>
    <note>Available at <uri>https://books.google.com/books/about
        /The_Python_Language_Reference_Manual.html?id=Ut4BuQAACAAJ</uri></note>
  </book>
  <article>
    <author>John K. Ousterhout</author>
    <title>Scripting: Higher-Level Programming for the 21st Century</title>
    <journal>Computer</journal>
    <volume>31</volume>
    <number>3</number>
    <month>March</month>
    <year>1998</year>
    <pages>23&#8211;30</pages>
  </article>
  <inproceedings>
    <author>Theodor Holm Nelson</author>
    <title>Complex Information Processing: A File Structure for the
        Complex, the Changing, and the Indeterminate</title>
    <booktitle>Proceedings of the Twentieth ACM National Conference</booktitle>
    <month>August</month>
    <year>1965</year>
    <address>Cleveland, OH</address>
    <pages>84&#8211;100</pages>
  </inproceedings>
  <inproceedings>
    <author>Stephan Kepser</author>
    <title>A Simple Proof for the Turing-Completeness of XSLT and XQuery</title>
    <booktitle>Proceedings, Extreme Markup Languages 2004</booktitle>
    <address>Montr&#233;al, Canada</address>
    <year>2004</year>
    <month>August</month>
    <note>Available at <uri>https://citeseerx.ist.psu.edu/document?
        doi=5f7ad1d9c17c01e3321b44ad996ff3fcd3ddbea3</uri></note>
  </inproceedings>
```

**Figure 14.22** A bibliography in XML. References (two books, a journal article, and three conference papers) appear in arbitrary order. The two URIs have been wrapped to fit on the printed page. *(continued)*

```
  <inproceedings>
    <author>David G. Korn</author>
    <title><code>ksh</code>: An Extensible High Level Language</title>
    <booktitle>Proceedings of the USENIX Very High Level Languages Symposium</booktitle>
    <address>Santa Fe, NM</address>
    <year>1994</year>
    <month>October</month>
    <pages>129&#8211;146</pages>
  </inproceedings>
  <book>
    <author>Tom Christiansen</author>
    <author>brian d foy</author>
    <author>Larry Wall</author>
    <author>Jon Orwant</author>
    <title>Programming Perl</title>
    <edition>fourth</edition>
    <publisher>O&#8217;Reilly Media</publisher>
    <address>Sebastopol, CA</address>
    <year>2012</year>
  </book>
</bibliography>
```

Figure 14.22 *(continued)*

document intended for the Web into a multipage document intended for printing on paper.

XQuery is a language in which to frame information-retrieval questions for a database stored in XML format. (In a bibliographic database, for example, we might use XQuery look for journal articles written since the turn of the century.) The purpose and behavior of XQuery parallel those of SQL, the standard language used for relational database queries. For the sake of simplicity, we will not use XSL-FO or XQuery in our extended example. Rather we will peruse an entire XML document, using XSLT to format its content as HTML.

An XML document can explicitly specify an XSLT script that should be used to transform or format it. This is a common but somewhat restrictive way to go about things: by tying a single stylesheet to the XML file we compromise the separation between content and presentation that was a principal motivation for creating XML in the first place. An alternative is to use client-side JavaScript or server-side PHP to invoke the XSLT processor, passing the XML document and the XSLT script as arguments. As of 2023, the XSLT 3 is the newest version of the language. XSLT 1 support is included in all major browsers; newer versions typically require a JavaScript library.

#### *Extended Example: Bibliographic Formatting*

**EXAMPLE 14.89**
Creating a reference list with XSLT

As an example of a task for which we might realistically use XSLT, consider the creation of a bibliographic reference list. Figure C-14.22 contains XML source for

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="https://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
  <html><head><title>Bibliography</title></head><body><h1>Bibliography</h1><ol>
    <xsl:for-each select="bibliography/*"><xsl:sort select="title"/>
      <li><xsl:apply-templates select="."/></li>
    </xsl:for-each>
  </ol></body></html>
</xsl:template>

<xsl:template match="bibliography/article">
  <q><xsl:apply-templates select="title/node()"/>,</q>
  by <xsl:call-template name="author-list"/>. 
  <em><xsl:apply-templates select="journal/node()"/>
  <xsl:text> </xsl:text><xsl:apply-templates select="volume/node()"/>
  </em>:<xsl:apply-templates select="number/node()"/>
  (<xsl:apply-templates select="month/node()"/><xsl:text> </xsl:text>
    <xsl:apply-templates select="year/node()"/>),
  pages <xsl:apply-templates select="pages/node()"/>.
  <xsl:if test="note"><xsl:apply-templates select="note/node()"/>.</xsl:if>
</xsl:template>

<xsl:template match="bibliography/book">
  <em><xsl:apply-templates select="title/node()"/>,</em>
  by <xsl:call-template name="author-list"/>. 
  <xsl:apply-templates select="publisher/node()"/>,
  <xsl:apply-templates select="address/node()"/>,
  <xsl:if test="edition">
    <xsl:apply-templates select="edition/node()"/> edition, </xsl:if>
  <xsl:apply-templates select="year/node()"/>.
  <xsl:if test="note"><xsl:apply-templates select="note/node()"/>.</xsl:if>
</xsl:template>

<xsl:template match="bibliography/inproceedings">
  <q><xsl:apply-templates select="title/node()"/>,</q>
  by <xsl:call-template name="author-list"/>. 
  In <em><xsl:apply-templates select="booktitle/node()"/></em>
  <xsl:if test="pages">, pages <xsl:apply-templates select="pages/node()"/></xsl:if>
  <xsl:if test="address">, <xsl:apply-templates select="address/node()"/></xsl:if>
  <xsl:if test="month">, <xsl:apply-templates select="month/node()"/></xsl:if>
  <xsl:if test="year">, <xsl:apply-templates select="year/node()"/></xsl:if>.
  <xsl:if test="note"><xsl:apply-templates select="note/node()"/>.</xsl:if>
</xsl:template>
```

**Figure 14.23** **Bibliography stylesheet in XSL.** This script will generate HTML when applied to a bibliography like that of Figure C-14.22. *(continued)*

```
<xsl:template name="author-list">       <!-- format author list -->
  <xsl:for-each select="author|editor">
    <xsl:if test="last() > 1 and position() = last()"> and </xsl:if>
    <xsl:apply-templates select="./node()"/>
    <xsl:if test="self::editor"> (editor)</xsl:if>
    <xsl:if test="last() > 2 and last() > position()">, </xsl:if>
  </xsl:for-each>
</xsl:template>

<xsl:template match="uri">              <!-- format link -->
    <a><xsl:attribute name="href"><xsl:value-of select="."/></xsl:attribute>
    <xsl:value-of select="substring-after(., 'https://')"/></a>
</xsl:template>

<xsl:template match="@*|node()">        <!-- default: copy content -->
  <xsl:copy><xsl:apply-templates select="@*|node()"/></xsl:copy>
</xsl:template>

</xsl:stylesheet>
```

Figure 14.23 *(continued)*

such a list. (Field names have been borrowed from BibTeX [Lam94, App. B].) The document begins with a declaration to specify the XML version and character encoding, and a processing instruction to specify the XSL stylesheet to be used to format the file. These declarations are included for the benefit of tools that process the document; they aren't part of the XML source itself. (Note the syntactic resemblance to the *processing instructions* used in Section C-14.3.2 to provide input to the PHP interpreter.)

At the top level, the `bibliography` element consists of a series of `book`, `article`, and `inproceedings` elements, each of which may contain elements for author and editor names, title, publisher, date and address, and so on. Some elements may contain nested `uri` elements, which specify on-line links. Characters that cannot be represented in ASCII are shown as Unicode *character entities*, as described in Sidebar 7.3.

Figure C-14.23 contains an XSLT stylesheet (script) to format the bibliography as HTML, which may then be rendered in a browser. This script was named at the beginning of the XML document (Figure C-14.22). In a manner analogous to that of the XML document, the script begins with a declaration to specify the XML version and character encoding, and an `xsl:stylesheet` element to specify the XSL version and namespace. The remainder of the script contains a mix of XSL and HTML elements. The XSL tags all specify the `xsl:` namespace explicitly. They are recognized by the XSLT processor. Elements from other namespaces are treated as ordinary text, to be copied through to the output when encountered.

The fundamental construct in XSLT is the `template`, which specifies a set of *instructions* to be applied to nodes in an XML source tree. Templates are typically

invoked by executing an `apply-templates` or a `call-template` instruction in some other template. Each invocation has a concept of *current node*. The execution as a whole begins by invoking an initial template with the root of the source tree (`/`) as current node. In our bibliographic example, the initial template is the one at the top of the script, because its `match` attribute is the XPath expression `"/"`. The body of the initial template begins with a string of HTML elements and text. This string is copied directly to the output. The `for-each` element, however, is an XSLT instruction, so it is executed.

The `select` attribute of the `for-each` element uses an XPath expression (`"bibliography/*"`) to build a *node set* consisting of all top-level entries in our bibliography. Other expressions could have been used if we wanted to be selective: `"bibliography/*[year>=2000]"` would match only recent entries; `"bibliography/*[note]"` would match only entries with `note` elements; `"bibliography/article|bibliography/book"` would match only articles and books.

The nested `sort` instruction forces the selected node set to be ordered alphabetically by title. The body of the `for-each` is then executed with each entry in turn selected as current node. The body contains a recursive invocation of `apply-templates`, bracketed by HTML list tags (`<li>` ... `</li>`). These tags are copied to the output, with the result of the recursive call nested in between.

So how does the recursive call work? Its `select` attribute, like that of `for-each`, uses XPath to build a node set. In this case it is the trivial node set containing only `"."`, the current node of the current iteration of `for-each`. The XSLT processor searches for a template that matches this node. We have created three appropriate candidates, one for each kind of bibliographic entry. When it finds the matching template, the processor invokes it, with an updated notion of current node.

Each of our three main templates contains a set of instructions to format its kind of entry (article, book, conference paper). Most of the instructions use additional invocations of `apply-templates` to format individual portions of an entry (author, title, publisher, etc.). Interspersed in these instructions are snippets of text and HTML elements. In several cases we use an `if` instruction to generate output only when a given XML element is present in the source. In most of these the recursive call uses the XPath `node()` function to select all children of the element in question.

White space is ignored when it comes between the end of one instruction and the beginning of the next. To force white space into the output in this case, we must delimit it with `<text>` ... `</text>` tags. Extra white space (e.g., after the ends of sentences) is specified with the "nonbreaking space" character entity, ` `.

Three extra templates end our script. The most interesting of these serves to format author lists. It has a `name` attribute rather than a `match` attribute, and is invoked with `call-template` rather than `apply-templates`. A `call`ed template always takes the current node of the caller—in this case the node that represents a bibliographic entry. Internally, the author list template executes a `for-each` instruction that selects all child nodes representing authors or editors. The `for-each`, in turn, uses the XPath `last()` and `position()` functions to determine how many

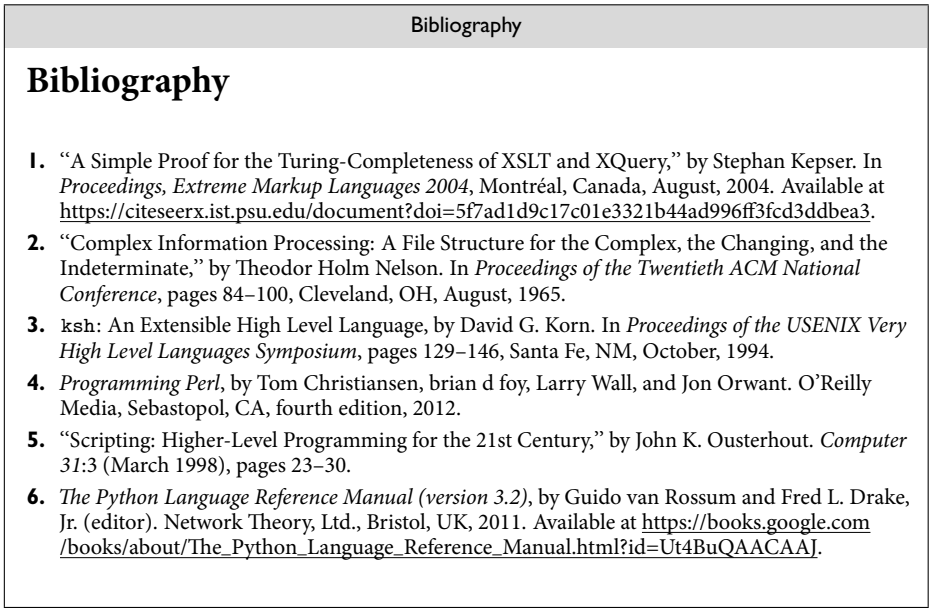
```
<html><head><title>Bibliography</title></head>
<body><h1>Bibliography</h1><ol>
<li>
  <q>A Simple Proof for the Turing-Completeness of XSLT and XQuery,</q>
  by Stephan Kepser.  In <em>Proceedings, Extreme Markup Languages
  2004</em>, Montr&eacute;al, Canada, August, 2004.  Available at
  <a href="https://citeseerx.ist.psu.edu/document?doi=
5f7ad1d9c17c01e3321b44ad996ff3fcd3ddbea3">citeseerx.ist.psu.edu
/document?doi=5f7ad1d9c17c01e3321b44ad996ff3fcd3ddbea3</a>.</li>
<li>
  <q>Complex Information Processing: A File Structure for the Complex,
  the Changing, and the Indeterminate,</q> by Theodor Holm Nelson. 
  In <em>Proceedings of the Twentieth ACM National Conference</em>,
  pages 84&ndash;100, Cleveland, OH, August, 1965.</li>
<li>
  <q><code>ksh</code>: An Extensible High Level Language,</q> by David
  G. Korn.  In <em>Proceedings of the USENIX Very High Level Languages
  Symposium</em>, pages 129&ndash;146, Santa Fe, NM, October, 1994.</li>
<li>
  <em>Programming Perl,</em> by Tom Christiansen, brian d foy, Larry Wall,
  and Jon Orwant.  O&rsquo;Reilly Media, Sebastopol, CA, fourth
  edition, 2012.</li>
<li>
  <q>Scripting: Higher-Level Programming for the 21st Century,</q> by
  John K. Ousterhout.  <em>Computer 31</em>:3 (March 1998), pages
  23&ndash;30.</li>
<li>
  <em>The Python Language Reference Manual (version 3.2),</em> by Guido
  van Rossum and Fred L. Drake, Jr. (editor).  Network Theory, Ltd.,
  Bristol, UK, 2011.  Available at <a href="https://books.google.com/books/about
/The_Python_Language_Reference_Manual.html?id=Ut4BuQAACAAJ">books.google.com/books
/about/The_Python_Language_Reference_Manual.html?id=Ut4BuQAACAAJ</a>.</li>
</ol>
</body></html>
```

Figure 14.24 Result of applying the stylesheet of Figure C-14.23 to the bibliography of Figure C-14.22.

names there are, and where each name falls in the list. It inserts the word "and" between the final two names, and puts commas after all names but the last in lists of three or more.

The template with `match="uri"` serves to format URIs that appear anywhere in the XML source. It creates an HTML link in the output, but uses the XPath `substring-after` function to strip the leading *https://* off the visible text. XPath provides a variety of similar functions for string and regular expression manipulation. The `value-of` instruction copies the contents of the selected node to the output, as a character string.

Our final template serves as a default case. The XPath expression `"@*|node()"` will match any attribute or other node in the XML source. Inside, the `copy` instruc-


Bibliography

## Bibliography

1. "A Simple Proof for the Turing-Completeness of XSLT and XQuery," by Stephan Kepser. In *Proceedings, Extreme Markup Languages 2004*, Montréal, Canada, August, 2004. Available at https://citeseerx.ist.psu.edu/document?doi=5f7ad1d9c17c01e3321b44ad996ff3fcd3ddbea3.
2. "Complex Information Processing: A File Structure for the Complex, the Changing, and the Indeterminate," by Theodor Holm Nelson. In *Proceedings of the Twentieth ACM National Conference*, pages 84–100, Cleveland, OH, August, 1965.
3. `ksh`: An Extensible High Level Language, by David G. Korn. In *Proceedings of the USENIX Very High Level Languages Symposium*, pages 129–146, Santa Fe, NM, October, 1994.
4. *Programming Perl*, by Tom Christiansen, brian d foy, Larry Wall, and Jon Orwant. O'Reilly Media, Sebastopol, CA, fourth edition, 2012.
5. "Scripting: Higher-Level Programming for the 21st Century," by John K. Ousterhout. *Computer 31*:3 (March 1998), pages 23–30.
6. *The Python Language Reference Manual (version 3.2)*, by Guido van Rossum and Fred L. Drake, Jr. (editor). Network Theory, Ltd., Bristol, UK, 2011. Available at https://books.google.com/books/about/The_Python_Language_Reference_Manual.html?id=Ut4BuQAACAAJ.

Figure 14.25 Rendered version of the HTML in Figure C-14.24.

tion copies the node's tags, if any, to the output, with the result of a recursive call to `apply-templates` in between. The `"@*|node()"` on the recursive call selects a node set consisting of all the current node's attributes and children. The end result is that any XML elements in the source that are delimited by tags for which we do not have special templates will be regenerated in the output just as they appear in the source. The recursion stops at text nodes and attributes, which are the leaves of the XML tree.

HTML output from our script appears in Figure C-14.24. The rendered web page appears in Figure C-14.25.

While lengthy by the standards of this text, our example illustrates only a fraction of the capabilities of XSLT. In the standard categorization of programming languages, the notation is strongly declarative: values may have names, but there are no mutable variables, and no side effects. There is a limited looping mechanism (`for-each`), but the real power comes from recursion, and from recursive traversal of XML trees in particular.

### CHECK YOUR UNDERSTANDING

56. Explain the relationships among SGML, HTML, and XML. What are their corresponding stylesheet languages?

57. Why does XML work so hard to distinguish between *content* and *presentation*?

58. What are the four main components of XSL? What are their respective purposes?
59. What is XHTML? How does it differ from "ordinary" HTML?
60. Explain the correspondence between XML documents and trees.
61. What does it mean for an XML document to be *well formed*?
62. Explain the distinctions (syntactic and semantic) among *elements*, *declarations*, and *processing instructions* in XML. Also explain the distinctions among *elements*, *tags*, and *attributes*.
63. Summarize the execution model of XSLT. In a nutshell, how does it work?
64. Explain the difference between *applying* templates and *calling* them in XSLT.

# 14 Scripting

## 14.6 Exercises

**14.15** Explain the circumstances under which it makes sense to realize an interactive task on the Web as a CGI script, an embedded server-side script, or a client-side script. For each of these implementation choices, give three examples of tasks for which it is clearly the preferred approach.

**14.16** (a) Write a web page with embedded PHP to print the first 10 rows of Pascal's triangle (see Example C-17.10 if you don't know what this is). When rendered, your output should look like Figure C-14.26.

(b) Modify your page to create a self-posting form that accepts the number of desired rows in an input field.

(c) Rewrite your page in JavaScript.

**14.17** Create a fill-in web form that uses a JavaScript implementation of the Luhn formula (Exercise C-4.27) to check for typos in credit card numbers. (But don't use real credit card numbers; homework exercises don't tend to be very secure!)

**14.18** (a) Modify the code of Figure C-14.20 (Example C-14.83) so that it replaces the form with its output, as the CGI and PHP versions of Figures C-14.16 and C-14.19 do.

(b) Modify the CGI and PHP scripts of Figures C-14.16 and C-14.19 (Examples C-14.78 and C-14.82) so they appear to append their output to the bottom of the form, as the JavaScript version of Figure C-14.20 does.

**14.19** Modify the XSLT of Figure C-14.23 to do one or more of the following:

(a) Alter the titles of conference papers so that only first words, words that follow a dash or colon (and thus begin a subtitle), and proper nouns are capitalized. You will need to adopt a convention by which the creator of the document can identify proper nouns.

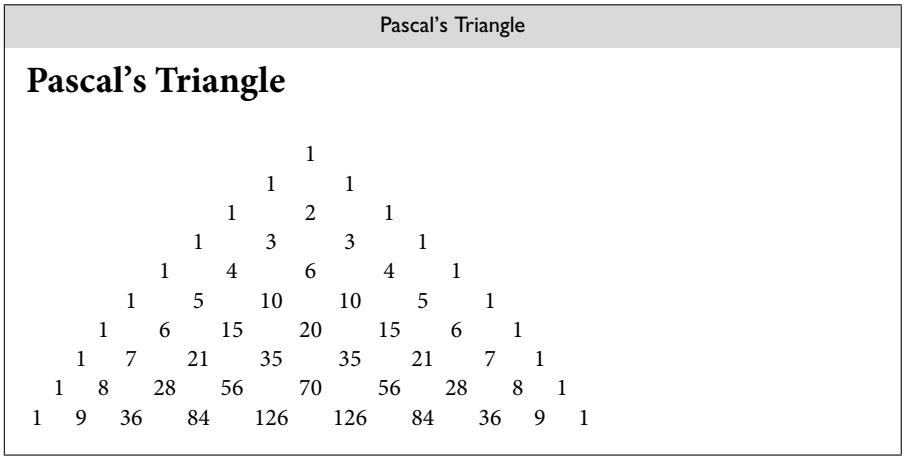



Figure 14.26 Pascal's triangle rendered in a web page (Exercise C-14.16).

(b) Sort entries by the last name of the first author or editor. You will need to adopt a convention by which the creator of the document can identify compound last names ("von Neumann," for example, should be alphabetized under 'v').

(c) Allow bibliographic entries to contain an `abstract` element, which when formatted appears as an indented block of text in a smaller font.

(d) In addition to the `book`, `article`, and `inproceedings` elements, add support for other kinds of entries, such as manuals, technical reports, theses, newspaper articles, web sites, and so on. You may want to draw inspiration from the categories supported by BibTeX [Lam94, App. B].

(e) Format entries according to some standard style convention (e.g., that of the Chicago Manual of Style [*chicagomanualofstyle.org/book/ed17/part3/ch14/toc.html*] or the ACM Transactions [*acm.org/publications/authors/submissions*]).

**14.20** Suppose bibliographic entries in Figure C-14.22 contain a mandatory `key` element, and that other documents can contain matching `cite` elements. Create an XSLT script that imitates the work of BibTeX. Your script should

(a) read an XML document, find all the `cite` elements, collect the keys they contain, and replace them with `bibref` elements that contain small integers instead.

(b) read a separate XML bibliography document, extract the entries with matching keys, and write them, in sorted order, to a new (and probably smaller) bibliography.

The small numbers in the `bibref` elements of the new document from (a) should match the corresponding numbered entries in the new bibliography from (b).

**14.21** Write a program that will read an XHTML file and print an outline of its contents, by extracting all `<title>`, `<h1>`, `<h2>`, and `<h3>` elements, and printing them at varying levels of indentation. Write

(a) in C or Java

(b) in `sed` or `awk`

(c) in Perl, Python, or Ruby

(d) in XSLT

Compare and contrast your solutions.

# 14 Scripting

## 14.7 Explorations

**14.32** Learn about Dart, a language developed at Google. Initially intended as a successor to JavaScript, Dart is now supported only as a language in which to develop code that will be *translated into* JavaScript. What explains the change in strategy?

**14.33** Learn more about WebAssembly. Why has it been successful when previous proposed alternatives to JavaScript were not?

**14.34** Learn more about DTDs and XML Schemas. Compare the DTD and XML Schema definitions of XHTML. What appear to the prospects for migrating to the newer specification language?

**14.35** Academics often keep lists of publications in multiple places and formats: an on-line web page, a printable resume, a BibTeX database for paper writing [Lam94, App. B]. Using XSLT, build a set of tools that will construct these lists automatically from a single XML source file.

**14.36** Learn about XSL-FO. Use it to reimplement Example C-14.89. Your new version should be a two-stage process: one XSLT script should add formatting tags to the XML bibliography; a second should convert the tagged bibliography to XHTML. Try to make these stages as general as possible: you should be able to modify the appearance of the output list by changing the first script only. You should also be able to write alternative versions of the second script that generate output in formats other than XHTML (e.g., LaTeX).

**14.37** Learn more about the history of W3C and WHATWG. What are the comparative advantages and disadvantages of their approaches to standardization? Do you find yourself more in sympathy with one approach or the other? How large are the technical differences between the most recent versions of the HTML standards? Are these differences significant enough to pose a problem for web developers?