

13

Scripting Languages

13.3.5 XSLT

HTML was inspired by an older standard known as SGML (standard generalized markup language), widely used in the business world to represent structured data. Because it evolved in such an ad hoc way, HTML has been very difficult to standardize. Incompatibilities among browsers continue to frustrate web designers, and several features of the language that have been deprecated 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 13.83

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 frequently use `<I> ... </I>` tags to request that text be set in an italic font, when ` ... ` (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 more and more vendors work to bring web content to cell phones, televisions, handheld computers, and audio-only devices, the need to distinguish between content and presentation is becoming increasingly critical. SGML has always made this distinction, but it is widely seen as overkill: far too complex for use on the web. ■

This is where XML steps in. XML (extensible markup language) is a deliberately streamlined descendant of SGML with at least three important advantages over HTML: (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). As noted in the main text, 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. It helps to remember the three families of markup languages—SGML, HTML, and XML—and to know that each 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 on stylesheets, but web sites increasingly use CSS to create a uniform “look and feel” across a collection of pages without embedding redundant information in every page.

SGML and DSSSL remain important in the business world, but are little used on the web. HTML is likely to persist for a very long time, but its lack of extensibility and its mix of content and presentation are increasingly perceived as fundamental limitations. XML is widely viewed as the notation of the future. Even for documents that remain in HTML, designers are likely to migrate toward XHTML (extensible hypertext markup language), an almost (but not quite) backward compatible variant of HTML that conforms to the XML standard.

XML and XHTML

An XML document must be *well formed*: tags must either constitute properly nested, matched pairs, or be explicit singletons, which end with a “/>” delimiter. The following fragment, for example, is well-formed (though incomplete) XHTML:

EXAMPLE 13.84

Well-formed XHTML

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

Here the quotation element (<q> ... </q>) is nested inside the emphasis element (...). Moreover the anchor element (<a ... />), which can serve as the target of a link, 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, or if the opening <q> tags were reversed (<q>). ■

Well-formedness is a simple syntactic rule, like the requirement that parentheses be balanced in Lisp. It makes XML (and thus XHTML) much easier than plain HTML to parse and to process automatically. The careful reader may also have noticed that we used lower-case letters for tags in XHTML, where previous HTML examples were all in upper case. HTML is case-insensitive; either style is accepted, though upper case has been the convention in standards documents. XML is case-sensitive, so and are different. The XHTML designers had to pick one. Going against the existing convention (but not the existing rules) preserves backward compatibility while helping the reader identify documents that are likely to conform to the newer standard.

The set of tags to be used in an XML document is specified by either a *document type definition* (DTD) or an *XML Schema*. DTDs are inherited from SGML. They indicate which tags are allowed, whether they are pairs or singletons, whether they permit attributes (name-value pairs like the `id="favorite-quote"` in Example ©13.84), and whether any attributes are mandatory. The rules of the DTD take the form of XML *declarations*, which look like elements beginning with a “<!” delimiter. These can be included directly in the XML document. More often they are kept in an external document with its own URI, and the XML document begins with a `<!DOCTYPE ... >` declaration that specifies that URI. (Comments also look like declarations: `<!-- ignored -->`.) If an XML document has no explicit DTD (neither in-line nor external), it is said to define a DTD *implicitly* by virtue of which tags are actually used.

XML Schemas are a newer mechanism, meant to replace DTDs. They are written in XSD, the XML Schema Definition language, which is itself an example of well-formed XML, defined by a DTD. Because they are written in XSD, XML Schemas can be created using XML-aware editors, parsed with XML parsers, and transformed with XSLT. In comparison to DTDs, XSD provides a significantly richer vocabulary for specifying syntactic rules. Among other things, it allows the designer to specify the data types of elements and attributes in considerable detail, providing a level of automatic checking not possible with DTDs. XSD also supports inheritance, so one XML Schema can be defined as an extension of another. As of this writing, DTDs remain more common than XML Schemas. In particular, the XML Schema for XHTML did not become official until 2008. We will rely on DTDs in the remainder of this section.

EXAMPLE 13.85
XHTML to display a
favorite quote

Because tags must nest in XML, a document has a natural tree-based structure. Figure ©13.24 shows the source for a small but complete XHTML 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 (non-leaf) 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.

Our document begins with an `<?xml ... ?>` declaration, which indicates the version of XML and the character encoding used in the rest of the document. The declaration is included for the benefit of tools that process the document; it isn’t part of the XML source itself. (Note the syntactic resemblance to the *processing instructions* used in Section 13.3.2 to provide input to the PHP interpreter.)

The second line of our document is a `<!DOCTYPE ... >` declaration that names an XHTML DTD at the World Wide Web Consortium. The remainder of the document is data. The root, named “/”, has one child: the `html` element. This in turn has two children: the `head` and the `body`. The `head` has a `title` child and an `xmlns` attribute. The latter declares `xhtml` to be the default *namespace* for the document. Namespaces in XML are similar to the namespaces of C++ or the packages of Java (Section 3.8); they allow us to give tag names a disambiguating 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. ■

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Favorite Quote</title>
</head>
<body>
<p>
<em><q><a id="favorite-quote" />
I defy the tyranny of precedent</q>
(Clara Barton).</em>
</p>
</body>
</html>
```

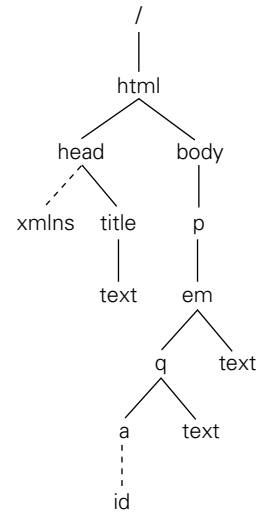


Figure 13.24 A complete XHTML document and its corresponding tree. Child relationships are shown with solid lines, attributes with dashed lines.

XSLT, XPath, and XSL-FO

XSL (extensible stylesheet language) can be thought of as a language for specifying what to *do* with an XML document. It has three sublanguages, called XSLT, XPath, and XSL-FO. 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.

XPath is a language used to name things in XML files. XPath names frequently appear in the attributes of XSLT elements. Returning to Figure ©13.24, the quotation element of our document could be named in XPath as `/html/body/p/em/q`. The quotation element and its text-node sibling, together, could be named as `/html/body/p/em/*`. 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 document intended for the web into a multipage document intended for printing on paper. For the sake of simplicity, we will not use XSL-FO in any of our examples. Rather we will format XML documents by using XSLT to turn them into HTML.

EXAMPLE 13.86

XPath names for XHTML elements

```

<?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</title>
    <publisher>Network Theory, Ltd.</publisher>
    <address>Bristol, UK</address>
    <year>2003</year>
    <note>Available at <uri>http://www.network-theory.co.uk/docs/pylang/</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>http://www.mulberrytech.com/Extreme/Proceedings/html
      /2004/Kepser01/EML2004Kepser01.html</uri></note>
  </inproceedings>

```

Figure 13.25 A bibliography in XML. References (two books, a journal article, and three conference papers) appear in arbitrary order. The Kepser URI has 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>Larry Wall</author>
  <author>Tom Christiansen</author>
  <author>Jon Orwant</author>
  <title>Programming Perl</title>
  <edition>third</edition>
  <publisher>O&#8217;Reilly and Associates</publisher>
  <address>Cambridge, MA</address>
  <year>2000</year>
</book>
</bibliography>
```

Figure 13.25 (continued)

An XML document can explicitly specify an XSLT script that should be used to transform or format it. This is a standard 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. Unfortunately, as of this writing the details vary across both server and client platforms.

Extended Example: Bibliographic Formatting

EXAMPLE 13.87

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 ©13.25 contains XML source for such a list. (Field names have been borrowed from BIB_TEX [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.

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 the sidebar on page 295.

```

<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://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"/>.&#160;
  <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"/>.&#160;
  <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"/>.&#160;
  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 13.26 Bibliography stylesheet in XSL. This script will generate HTML when applied to a bibliography like that of Figure  13.25. (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(., 'http://')"/></a>
</xsl:template>

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

</xsl:stylesheet>
```

Figure 13.26 (continued)

Figure ©13.26 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 ©13.25). 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 `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` 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 (` ... `). 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 `called` 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 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 `http://` 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` instruction 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

```
<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.&nbsp;&nbsp;&nbsp; In <em>Proceedings, Extreme Markup Languages
  2004</em>, Montr&eacute;al, Canada, August, 2004. Available at
  <a href="http://www.mulberrytech.com/Extreme/Proceedings/html/2004/Kepser01
  /EML2004Kepser01.html">www.mulberrytech.com/Extreme/Proceedings/html/2004
  /Kepser01/EML2004Kepser01.html</a>.</li>
<li>
  <q>Complex Information Processing: A File Structure for the Complex,
  the Changing, and the Indeterminate,</q> by Theodor Holm Nelson.&nbsp;&nbsp;&nbsp;
  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.&nbsp;&nbsp;&nbsp; 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 Larry Wall, Tom Christiansen, and Jon
  Orwant.&nbsp;&nbsp;&nbsp; O&rsquo;Reilly and Associates, Cambridge, MA, third edition,
  2000.</li>
<li>
  <q>Scripting: Higher-Level Programming for the 21st Century,</q> by
  John K. Ousterhout.&nbsp;&nbsp;&nbsp; <em>Computer 31</em>:3 (March 1998), pages
  23&ndash;30.</li>
<li>
  <em>The Python Language Reference Manual,</em> by Guido van Rossum and
  Fred L. Drake, Jr. (editor).&nbsp;&nbsp;&nbsp; Network Theory, Ltd., Bristol, UK, 2003.
  Available at <a href="http://www.network-theory.co.uk/docs/pylang/">www.network-
  theory.co.uk/docs/pylang/</a>.</li>
</ol>
</body></html>
```

Figure 13.27 Result of applying the stylesheet of Figure 13.26 to the bibliography of Figure 13.25.

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 13.27. The rendered web page appears in Figure 13.28.

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. ■

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 www.mulberrytech.com/Extreme/Proceedings/html/2004/Kepser01/EML2004Kepser01.html.
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 Larry Wall, Tom Christiansen, and Jon Orwant. O’Reilly and Associates, Cambridge, MA, third edition, 2000.
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*, by Guido van Rossum and Fred L. Drake, Jr. (editor). Network Theory, Ltd., Bristol, UK, 2003. Available at www.network-theory.co.uk/docs/pylang/.

Figure 13.28 Rendered version of the HTML in Figure © 13.27.

✓ CHECK YOUR UNDERSTANDING

58. Explain the relationships among SGML, HTML, and XML. What are their corresponding stylesheet languages?
59. Why does XML work so hard to distinguish between *content* and *presentation*?
60. What are the three main components of XSL? What are their respective purposes?
61. What is XHTML? How does it differ from HTML?
62. Explain the correspondence between XML documents and trees.
63. What does it mean for an XML document to be *well formed*?
64. What is a *document type definition* (DTD)? An *XML Schema*? Briefly, how do they compare?
65. Explain the distinctions (syntactic and semantic) among *elements*, *declarations*, and *processing instructions* in XML. Also explain the distinctions among *elements*, *tags*, and *attributes*.
66. Summarize the execution model of XSLT. In a nutshell, how does it work?
67. Explain the difference between *applying* templates and *calling* them in XSLT.

