

XSL Transformations

The Extensible Style Language (XSL) includes both a transformation language and a formatting language. Each of these, naturally enough, is an XML application. The transformation language provides elements that define rules for how one XML document is transformed into another XML document. The transformed XML document may use the markup and DTD of the original document or it may use a completely different set of tags. In particular, it may use the tags defined by the second part of XSL, the formatting objects. This chapter covers the transformation language half of XSL.

What Is XSL?

The transformation and formatting halves of XSL can function independently of each other. For instance, the transformation language can transform an XML document into a well-formed HTML file, and completely ignore the XSL formatting objects. This is the style of XSL supported by Internet Explorer 5.0, previewed in Chapter 5, and emphasized in this chapter.

Furthermore, it's not absolutely required that a document written in XSL formatting objects be produced by using the transformation part of XSL on another XML document. For example, it's easy to imagine a converter written in Java that reads TeX or PDF files and translates them into XSL formatting objects (though no such converters exist as of May, 1999).

14

CHAPTER



In This Chapter

Understanding XSL, XSL transformations, and templates

Computing the value of a node

Processing multiple elements

Selecting nodes with Expressions

Understanding the default template rules

Deciding which output to include

Copying the current node

Counting nodes, sorting output elements, and inserting CDATA, and < signs

Setting the mode attribute

Defining and creating named templates

Stripping and preserving whitespace

Changing output based on input

Merging multiple style sheets



In essence, XSL is two languages, not one. The first language is a transformation language, the second a formatting language. The transformation language is useful independently of the formatting language. Its ability to move data from one XML representation to another makes it an important component of XML-based electronic commerce, electronic data interchange, metadata exchange, and any application that needs to convert between different XML representations of the same data. These uses are also united by their lack of concern with rendering data on a display for humans to read. They are purely about moving data from one computer system or program to another.

Consequently, many early implementations of XSL focus exclusively on the transformation part and ignore the formatting objects. These are incomplete implementations, but nonetheless useful. Not all data must ultimately be rendered on a computer monitor or printed on paper.

Cross-Reference

Chapter 15, *XSL Formatting Objects*, covers the XSL formatting language.

A Word of Caution about XSL

XSL is still under development. The XSL language has changed radically in the past, and will almost certainly change again in the future. This chapter is based on the April 21, 1999 (fourth) draft of the XSL specification. By the time you are reading this book, this draft of XSL will probably have been superseded and the exact syntax of XSL will have changed. I'm hopeful that this chapter won't be too far removed from the actual specification. Nonetheless, if you do encounter inconsistencies, you should compare the examples in this book against the most current specification.

To make matters worse, no software yet implements all of the April 21, 1999 (fourth) draft of the XSL specification, not even the transformation half. All products available now implement different subsets of the current draft. Furthermore, many products, including Internet Explorer 5.0 and XT add elements not actually present in the current draft specification for XSL. Finally, most products that attempt to implement at least part of XSL have non-trivial bugs in those parts they do implement. Consequently, very few examples will work exactly the same way in different software.

Eventually, of course, this should be straightened out as the standard evolves toward its final incarnation, as vendors fix the bugs in their products and implement the unimplemented parts, and as more software is published that supports XSL. Until then you're faced with a choice: you can either work out on the bleeding edge with XSL in its current, incomplete, unfinished state and try to work around all the bugs and omissions you'll encounter; or you can stick with a more established technology like CSS until XSL becomes more solid.

Overview of XSL Transformations

In an XSL transformation, an XSL processor reads both an XML document and an XSL style sheet. Based on the instructions the processor finds in the XSL style sheet, it outputs a new XML document.

Trees

As you learned in Chapter 6, every well-formed XML document is a tree. A tree is a data structure composed of connected nodes beginning with a single node called the root. The root connects to its child nodes, each of which may connect to zero or more children of its own, and so forth. Nodes that have no children of their own are called leaves. A diagram of tree looks much like a genealogical descendant chart that lists the descendants of a single ancestor. The most useful property of a tree is that each node and its children also form a tree. Thus, a tree is a hierarchical structure of trees in which each tree is built out of smaller trees.

The nodes of an XML tree are the elements and their content. However, for the purposes of XSL, attributes, namespaces, processing instructions, and comments must also be counted as nodes. Furthermore, the root of the document must be distinguished from the root element. Thus, XSL processors assume an XML tree contains seven kinds of nodes. These are:

1. The root
2. Elements
3. Text
4. Attributes
5. Namespaces
6. Processing instructions
7. Comments

For example, consider the XML document in Listing 14-1. This shows a periodic table of the elements that I'll use as an example in this chapter. (More properly it shows the first two elements of the periodic table.)



The complete periodic table appears on the CD-ROM in the file `allelements.xml` in the `examples/periodic_table` directory.

The root `PERIODIC_TABLE` element contains `ATOM` child elements. Each `ATOM` element houses a variety of child elements providing the atomic number, atomic weight, symbol, boiling point, and so forth. A `UNITS` attribute specifies the units for those elements that have units.

Note

ELEMENT would be a more appropriate choice here than ATOM. However, writing about ELEMENT elements and trying to distinguish between chemical elements and XML elements might create confusion. Thus, at least for the purposes of this chapter, ATOM seemed like the more legible choice.

Listing 14-1: An XML periodic table with two elements, hydrogen and helium

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="14-2.xsl"?>
<PERIODIC_TABLE>

  <ATOM STATE="GAS">
    <NAME>Hydrogen</NAME>
    <SYMBOL>H</SYMBOL>
    <ATOMIC_NUMBER>1</ATOMIC_NUMBER>
    <ATOMIC_WEIGHT>1.00794</ATOMIC_WEIGHT>
    <BOILING_POINT UNITS="Kelvin">20.28</BOILING_POINT>
    <MELTING_POINT UNITS="Kelvin">13.81</MELTING_POINT>
    <DENSITY UNITS="grams/cubic centimeter"><!-- At 300K -->
      0.0899
    </DENSITY>
  </ATOM>

  <ATOM STATE="GAS">
    <NAME>Helium</NAME>
    <SYMBOL>He</SYMBOL>
    <ATOMIC_NUMBER>2</ATOMIC_NUMBER>
    <ATOMIC_WEIGHT>4.0026</ATOMIC_WEIGHT>
    <BOILING_POINT UNITS="Kelvin">4.216</BOILING_POINT>
    <MELTING_POINT UNITS="Kelvin">0.95</MELTING_POINT>
    <DENSITY UNITS="grams/cubic centimeter"><!-- At 300K -->
      0.1785
    </DENSITY>
  </ATOM>

</PERIODIC_TABLE>
```

Figure 14-1 displays a diagram of this document as a tree. It begins at the top with the root node (not the same as the root element!) which contains two child nodes, the `xml-stylesheet` processing instruction and the root element `PERIODIC_TABLE`. (The XML declaration is not visible to the XSL processor and is not included in the tree the XSL processor operates on.) The `PERIODIC_TABLE` element contains two child nodes, both `ATOM` elements. Each `ATOM` element has an attribute node for its `STATE` attribute, and a variety of child element nodes. Each child element

encompasses a node for its contents, as well as nodes for any attributes and comments it possesses. Notice in particular that many nodes are something other than elements. There are nodes for text, attributes, comments, and processing instructions. Unlike CSS1, XSL is not limited to working only with whole elements. It has a much more granular view of a document that enables you to base styles on comments, attributes, processing instructions, and more.

Note

Like the XML declaration, an internal DTD subset or DOCTYPE declaration is not part of the tree. However, it may have the effect of adding attribute nodes to some elements through `<!ATTLIST>` declarations that use `#FIXED` or default attribute values.

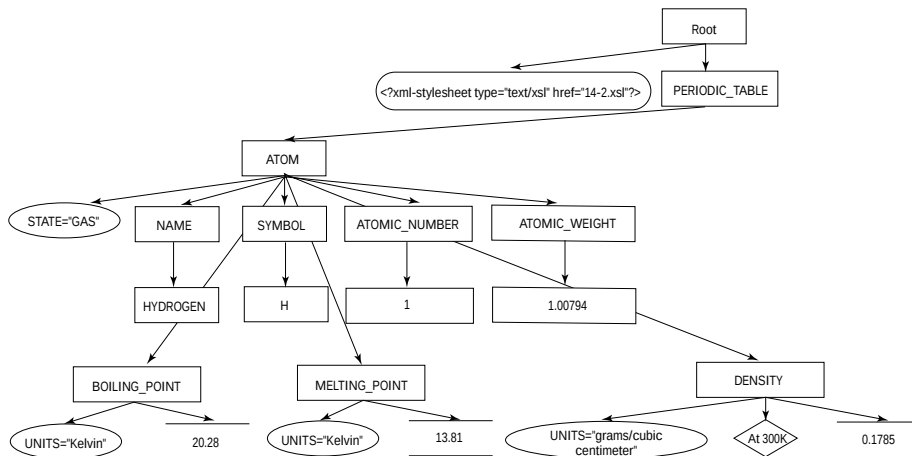


Figure 14-1: Listing 14-1 as a tree diagram

The XSL transformation language operates by transforming one XML tree into another XML tree. The language contains operators for selecting particular nodes from the tree, reordering the nodes, and outputting nodes. If one of these nodes is an element node, then it may be an entire tree itself. Remember that all these operators, both for input and output, are designed for operation on a tree. They are not a general regular expression language for transforming arbitrary data.

XSL Style Sheet Documents

More precisely, an XSL transformation accepts as input a tree represented as an XML document and produces as output a new tree, also represented as an XML document. Consequently, the transformation part of XSL is also called the tree construction part. Both the input and the output must be XML documents. You cannot use XSL to transform to or from non-XML formats like PDF, TeX, Microsoft Word, PostScript, MIDI, or

others. You can use XSL to transform XML to an intermediate format like TeXML, then use additional, non-XSL software to transform that into the format you want. HTML and SGML are borderline cases because they're so close to XML. You can use XSL to transform to or from HTML and SGML that meets XML's well-formedness rules. However, XSL cannot handle the wide variety of non-well-formed HTML and SGML you'll encounter on most Web sites and document production systems. The key thing to remember is that XSL transformation language works for XML-to-XML conversions, not for anything else.

An XSL document contains a list of template rules and other rules. A template rule has a pattern specifying the trees it applies to and a template to be output when the pattern is matched. When an XSL processor formats an XML document using an XSL style sheet, it scans the XML document tree looking through each sub-tree in turn. As each tree in the XML document is read, the processor compares it with the pattern of each template rule in the style sheet. When the processor finds a tree that matches a template rule's pattern, it outputs the rule's template. This template generally includes some markup, some new data, and some data copied out of the tree from the original XML document.

XSL uses XML to describe these rules, templates, and patterns. The XSL document itself is an `xsl:stylesheet` element. Each template rule is an `xsl:template` element. The pattern of the rule is the value of the `match` attribute of the `xsl:template` element. The output template is the content of the `xsl:template` element. All instructions in the template for doing things like selecting parts of the input tree to include in the output tree are performed by one or another XSL element. These are identified by the `xsl:` prefix on the element names. Elements that do not have an `xsl:` prefix are part of the result tree.



More properly, all elements that are XSL instructions are part of the `xsl` namespace. Namespaces are discussed in Chapter 18, *Namespaces*. Until then, all you have to know is that the names of all XSL elements begin with `xsl:`.

Listing 14-2 shows a very simple XSL style sheet with two template rules. The first template rule matches the root element `PERIODIC_TABLE`. It replaces this element with an `html` element. The contents of the `html` element are the results of applying the other templates in the document to the contents of the `PERIODIC_TABLE` element.

The second template matches `ATOM` elements. It replaces each `ATOM` element in the input document with a `P` element in the output document. The `xsl:apply-templates` rule inserts the text of the matched source element into the output document. Thus, the contents of a `P` element will be the text (but not the markup) contained in the corresponding `ATOM` element. I further discuss the exact syntax of these elements below.

Listing 14-2: An XSL style sheet for the periodic table with two template rules

```
<?xml version="1.0"?>
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/XSL/Transform/1.0">

  <xsl:template match="PERIODIC_TABLE">
    <html>
      <xsl:apply-templates/>
    </html>
  </xsl:template>

  <xsl:template match="ATOM">
    <P>
      <xsl:apply-templates/>
    </P>
  </xsl:template>

</xsl:stylesheet>
```

Where Does the XML Transformation Happen?

There are three primary ways XML documents are transformed into other formats, such as HTML, with an XSL style sheet:

1. The XML document and associated style sheet are both served to the client (Web browser), which then transforms the document as specified by the style sheet and presents it to the user.
2. The server applies an XSL style sheet to an XML document to transform it to some other format (generally HTML) and sends the transformed document to the client (Web browser).
3. A third program transforms the original XML document into some other format (often HTML) before the document is placed on the server. Both server and client only deal with the post-transform document.

Each of these three approaches uses different software, though they all use the same XML documents and XSL style sheets. An ordinary Web server sending XML documents to Internet Explorer 5.0 is an example of the first approach. A servlet-compatible Web server using the IBM alphaWorks' XML enabler exemplifies the second approach. Using the command line XT program to transform XML documents to HTML documents, then placing the HTML documents on a Web server is an example of the third approach. However, these all use (at least in theory) the same XSL language.

In this chapter, I will emphasize the third approach, primarily because at the time of this writing, specialized converter programs like James Clark's XT or the IBM's LotusXSL provide the most complete and accurate implementation of the current XSL specification. Furthermore, this provides the broadest compatibility with legacy Web browsers and servers whereas the first approach requires a more recent browser than most users use and the second approach requires special Web server software. In practice, though, requiring a different server is not nearly as onerous as requiring a particular client. You, yourself, can install your own special server software, but you cannot rely on your visitors to install particular client software.

How to Use XT

XT is a Java 1.1 character mode application. To use it, you'll need to have a Java 1.1 compatible virtual machine installed such as Sun's Java Development Kit (JDK) or Java Runtime Environment (JRE), Apple's Macintosh Runtime for Java 2.1 (MRJ), or Microsoft's virtual machine. You'll also need to install a SAX compliant XML parser like James Clark's XP. This is also a Java application.



Note

At the time of this writing, XT can be found at <http://www.jclark.com/xml/xt.html> and .XP can be found at <http://www.jclark.com/xml/xp/index.html>. These URLs are, of course, subject to change as time passes. Indeed, there's no guarantee that XT will be available when you read this. However, although I use XT in this chapter, the examples should work with any XSL processor that implements the tree construction part of the April 21, 1999 working draft of the XSL specification. Another possibility is IBM alphaWorks' LotusXSL, available at <http://www.alphaworks.ibm.com/tech/LotusXSL>. The examples may or may not work with software that implements later drafts of XSL, though I hope they'll be close. I'll post any updates on my own Web site at <http://metalab.unc.edu/xml/books/bible/>.

The Java class containing the main method for XT is `com.jclark.xml.sax.Driver`. Assuming your Java CLASSPATH environment variable includes the `xt.jar` and `sax.jar` files (both included in the XT distribution), you can run XT by typing the following at the shell prompt or in a DOS window:

```
C:\> java
-Dcom.jclark.xml.sax.parser=com.jclark.xml.sax.CommentDriver
com.jclark.xml.sax.Driver 14-1.xml 14-2.xml 14-3.html
```

This line runs the Java interpreter, sets the `com.jclark.xml.sax.parser` Java environment variable to `com.jclark.xml.sax.CommentDriver`, which indicates the fully qualified name of the Java class used to parse the input documents. This class must be somewhere in your class path. Here I've used the XP parser, but any SAX-compliant parser will do. Next comes the name of the Java class containing the XT program's `main()` method, `com.jclark.xml.sax.Driver`. Finally, there are the names of the input XML document (14-1.xml), the input XSL style sheet (14-2.xml), and the output HTML file (14-3.html). If the last argument is omitted, the transformed document will be printed on the console.

 Tip

If you're using Windows and have installed the Microsoft Java virtual machine, you can use a stand-alone executable version of XT instead. This is a little easier to use since it includes the XP parser and doesn't require you to mess around with CLASSPATH environment variables. With this program, you simply place the xt.exe file in your path, and type:

```
C:\> xt 14-1.xml 14-2.xsl 14-3.html
```

Listing 14-2 transforms input documents to well-formed HTML files as discussed in Chapter 6. However, you can transform from any XML application to any other as long as you can write a style sheet to support the transformation. For example, you can imagine a style sheet that transforms from VML documents to SVG documents:

```
% java
-Dcom.jclark.xml.sax.parser=com.jclark.xml.sax.CommentDriver
com.jclark.xml.sax.Driver pinktriangle.vml
VmlToSVG.xsl -out pinktriangle.svg
```

Most other command line XSL processors behave similarly, though of course they'll have different command line arguments and options. They may prove slightly easier to use if they're not written in Java since there won't be any need to configure the CLASSPATH.

Listing 14-3 shows the output of running Listing 14-1 through XT with the XSL style sheet in Listing 14-2. Notice that XT does not attempt to clean up the HTML it generates, which has a lot of whitespace. This is not important since ultimately you want to view the file in a Web browser that trims whitespace. Figure 14-2 shows Listing 14-3 loaded into Netscape Navigator 4.5. Since Listing 14-3 displays standard HTML, you don't need an XML-capable browser to view it.

Listing 14-3: The HTML produced by applying the style sheet in Listing 14-2 to the XML in Listing 14-1

```
<html>

  <P>
    Hydrogen
    H
    1
    1.00794
    20.28
    13.81

    0.0899

  </P>
```

Continued

Listing 14-3 (continued)

```
<P>
  Helium
  He
  2
  4.0026
  4.216
  0.95

  0.1785

</P>

</html>
```

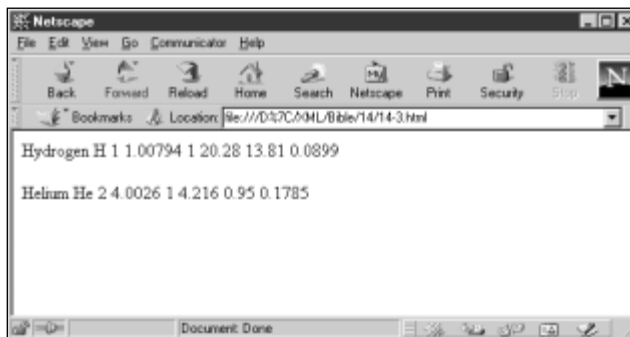


Figure 14-2: The page produced by applying the XSL style sheet in Listing 14-2 to the XML document in Listing 14-1

Direct Display of XML Files with XSL Style Sheets

Instead of preprocessing the XML file, you can send the client both the XML file, and the XSL file that describes how to render it. The client is responsible for applying the style sheet to the document and rendering it accordingly. This is more work for the client, but places much less load on the server. In this case, the XSL style sheet must transform the document into an XML application the client understands. HTML is a likely choice, though in the future some browsers will likely work with XSL formatting objects as well.

Attaching an XSL style sheet to an XML document is easy. Simply insert an `xml-stYLESHEET` processing instruction in the prolog immediately after the XML declaration. This processing instruction should have a `type` attribute with the value `text/xml` and an `href` attribute whose value is a URL pointing to the style sheet. For example:

```
<?xml version="1.0"?>
<?xml-stYLESHEET type="text/xml" href="14-2.xml"?>
```

This is also how you attach a CSS style sheet to a document. The only difference here is that the `type` attribute has the value `text/css` instead of `text/xml`.

Internet Explorer 5.0's XSL support differs from the 4-21-1999 working draft in a number of respects. First of all it expects that XSL elements are in the `http://www.w3.org/TR/WD-xsl` namespace instead of the `http://www.w3.org/XSL/Transform/1.0` namespace, although the `xsl` prefix is still used. Secondly, it does not implement the default rules for elements that match no template. Consequently, you need to provide a template for each element in the hierarchy starting from the root before trying to view a document in Internet Explorer. Listing 14-4 demonstrates. The three rules match the root node, the root element `PERIODIC_TABLE`, and the `ATOM` elements in that order. Figure 14-3 shows the XML document in Listing 14-1 loaded into Internet Explorer 5.0 with this style sheet.

Listing 14-4: The style sheet of Listing 14-2 adjusted to work with Internet Explorer 5.0

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">

  <xsl:template match="/">
    <html>
      <xsl:apply-templates/>
    </html>
  </xsl:template>

  <xsl:template match="PERIODIC_TABLE">
    <xsl:apply-templates/>
  </xsl:template>

  <xsl:template match="ATOM">
    <P>
      <xsl:value-of select="."/>
    </P>
  </xsl:template>

</xsl:stylesheet>
```

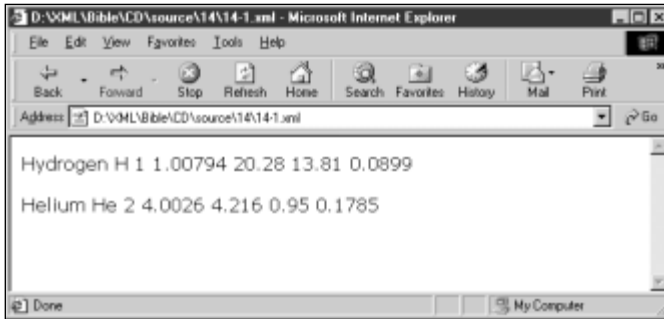


Figure 14-3: The page produced in Internet Explorer 5.0 by applying the adjusted XSL style sheet in Listing 14-4 to the XML document in Listing 14-1

Caution

Ideally, you would use the same XML document both for direct display and for pre-rendering to HTML. Unfortunately, XT won't accept the `http://www.w3.org/TR/WD-xsl` namespace and IE5 won't accept the `http://www.w3.org/XSL/Transform/1.0` namespace. Such is life on the bleeding edge as different processors play leapfrog in their support of various parts of the evolving XSL specification.

In the rest of this chapter, I will simply pre-render the file in HTML before loading it into a Web browser.

XSL Templates

Template rules defined by the `xsl:template` element are the most important part of the XSL style sheet. Each template rule is an `xsl:template` element. These associate particular output with particular input. Each `xsl:template` element has a `match` attribute that specifies which nodes of the input document the template is instantiated for.

The content of the `xsl:template` element is the actual template to be instantiated. A template may contain both text that will appear literally in the output document and XSL instructions that copy data from the input XML document to the result. Because all XSL instructions are in the `xsl` namespace (that is they all begin with `xsl:`), it's easy to distinguish between the elements that are literal data to be copied to the output and XSL instructions. For example, here is a template that is applied to the root node of the input tree:

```
<xsl:template match="/">
  <html>
    <head>
    </head>
    <body>
    </body>
```

```
</html>
</xsl:template>
```

When the XSL processor reads the input document, the first node it sees is the root. This rule matches that root node, and tells the XSL processor to emit this text:

```
<html>
  <head>
</head>
  <body>
</body>
</html>
```

This text is well-formed HTML. Since the XSL document is itself an XML document, its contents—templates included—must be well-formed XML.

If you were to use the above rule, and only the above rule, in an XSL style sheet, the output would be limited to the above six tags. (Actually they're compressed to the equivalent four tags `<html><head/><body/></html>`). That's because no instructions in the rule tell the formatter to move down the tree and look for further matches against the templates in the style sheet.

The `xsl:apply-templates` Element

To get beyond the root, you have to tell the formatting engine to process the children of the root. In general, to include content in the child nodes, you have to recursively process the nodes through the XML document. The element that does this is `xsl:apply-templates`. By including `xsl:apply-templates` in the output template, you tell the formatter to compare each child element of the matched source element against the templates in the style sheet; and, if a match is found, output the template for the matched node. The template for the matched node may itself contain `xsl:apply-templates` elements to search for matches for its children. When the formatting engine processes a node, the node is treated as a complete tree. This is the advantage of the tree structure. Each part can be treated the same way as the whole. For example, Listing 14-5 is an XSL style sheet that uses the `xsl:apply-templates` element to process the child nodes.

Listing 14-5: An XSL style sheet that recursively processes the children of the root

```
<?xml version="1.0"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/XSL/Transform/1.0">

  <xsl:template match="/">
    <html>
```

Continued

Listing 14-5: (continued)

```
<xsl:apply-templates/>
</html>
</xsl:template>

<xsl:template match="PERIODIC_TABLE">
  <body>
    <xsl:apply-templates/>
  </body>
</xsl:template>

<xsl:template match="ATOM">
  An Atom
</xsl:template>

</xsl:stylesheet>
```

When this style sheet is applied to Listing 14-1, here's what happens:

1. The root node is compared with all template rules in the style sheet. It matches the first one.
2. The `<html>` tag is written out.
3. The `xsl:apply-templates` element causes the formatting engine to process the child nodes.
 - A. The first child of the root, the `xml-stylesheet` processing instruction, is compared with the template rules. It doesn't match any of them so no output is generated.
 - B. The second child of the root, the root element `PERIODIC_TABLE`, is compared with the template rules. It matches the second template rule.
 - C. The `<body>` tag is written out.
 - D. The `xsl:apply-templates` element in the `body` element causes the formatting engine to process the child nodes of `PERIODIC_TABLE`.
 - a. The first child of the `PERIODIC_TABLE` element, that is the `Hydrogen ATOM` element, is compared with the template rules. It matches the third template rule.
 - b. The text "An Atom" is output.
 - c. The second child of the `PERIODIC_TABLE` element, that is the `Helium ATOM` element, is compared with the template rules. It matches the third template rule.

- d. The text “An Atom” is output.
- E. The `</body>` tag is written out.
- 4. The `</html>` tag is written out.
- 5. Processing is complete.

The end result is:

```
<html><body>

  An Atom

  An Atom

</body></html>
```

The select Attribute

To replace the text “An Atom” with the name of the `ATOM` element as given by its `NAME` child, you need to specify that templates should be applied to the `NAME` children of the `ATOM` element. To choose a particular set of children instead of all children you supply `xsl:apply-templates` with a `select` attribute designating the children to be selected. In this example:

```
<xsl:template match="ATOM">
  <xsl:apply-templates select="NAME"/>
</xsl:template>
```

The `select` attribute uses the same kind of patterns as the `match` attribute of the `xsl:template` element. For now, we’ll stick to simple names of elements; but in the section on patterns for matching and selecting later in this chapter, we’ll explore many more possibilities for both `select` and `match`. If no `select` attribute is present, all child elements are selected.

The result of adding this rule to the style sheet of Listing 14-5 and applying it to Listing 14-5 is this:

```
<html><head/><body>

  Hydrogen

  Helium

</body></html>
```

Computing the Value of a Node with `xsl:value-of`

The `xsl:value-of` element copies the value of a node in the input document into the output document. The `select` attribute of the `xsl:value-of` element specifies which node's value is being taken.

For example, suppose you want to replace the literal text “An Atom” with the name of the `ATOM` element as given by the contents of its `NAME` child. You can replace An Atom with `<xsl:value-of select="NAME"/>` like this:

```
<xsl:template match="ATOM">
  <xsl:value-of select="NAME"/>
</xsl:template>
```

Then, when you apply the style sheet to Listing 14-1, this text is generated:

```
<html><head/><body>

  Hydrogen

  Helium

</body></html>
```

The item whose value is selected, the `NAME` element in this example, is relative to the source node. The source node is the item matched by the template, the particular `ATOM` element in this example. Thus, when the Hydrogen `ATOM` is matched by `<xsl:template match="ATOM">`, the Hydrogen `ATOM`'s `NAME` is selected by `xsl:value-of`. When the Helium `ATOM` is matched by `<xsl:template match="ATOM">`, the Helium `ATOM`'s `NAME` is selected by `xsl:value-of`.

The value of a node is always a string, possibly an empty string. The exact contents of this string depend on the type of the node. The most common type of node is element, and the value of an element node is particularly simple. It's the concatenation of all the parsed character data (but not markup!) between the element's start tag and end tag. For example, the first `ATOM` element in Listing 14-1 is as follows:

```
<ATOM STATE="GAS">
  <NAME>Hydrogen</NAME>
  <SYMBOL>H</SYMBOL>
  <ATOMIC_NUMBER>1</ATOMIC_NUMBER>
  <ATOMIC_WEIGHT>1.00794</ATOMIC_WEIGHT>
  <OXIDATION_STATES>1</OXIDATION_STATES>
  <BOILING_POINT UNITS="Kelvin">20.28</BOILING_POINT>
  <MELTING_POINT UNITS="Kelvin">13.81</MELTING_POINT>
```

```

    <DENSITY UNITS="grams/cubic centimeter"><!-- At 300K -->
      0.0899
    </DENSITY>
  </ATOM>

```

The value of this element is shown below:

```

Hydrogen
H
1
1.00794
1
20.28
13.81

0.0899

```

I calculated that, just by stripping out all the tags and comments. Everything else including whitespace was left intact. The values of the other six node types are calculated similarly, mostly in obvious ways. Table 14-1 summarizes.

Table 14-1
Values of Nodes

<i>Node Type</i>	<i>Value</i>
Root	the value of the root element
Element	the concatenation of all parsed character data contained in the element, including character data in any of the descendants of the element
Text	the text of the node; essentially the node itself
Attribute	the normalized attribute value as specified by Section 3.3.3 of the XML 1.0 recommendation; basically the attribute value after entities are resolved and leading and trailing whitespace is stripped; does not include the name of the attribute, the equals sign, or the quotation marks
Namespace	the URI for the namespace
Processing instruction	the value of the processing instruction; does not include the processing instruction name, <? or ?>
Comment	the text of the comment, <!-- and --> not included

Processing Multiple Elements with `xsl:for-each`

The `xsl:value-of` element should only be used in contexts where it is unambiguous as to which node's value is being taken. If there are multiple possible items that could be selected, then only the first one will be chosen. For instance, this is a poor rule because a typical `PERIODIC_TABLE` element contains more than one `ATOM`:

```
<xsl:template match="PERIODIC_TABLE">
  <xsl:value-of select="ATOM"/>
</xsl:template>
```

There are two ways of processing multiple elements in turn. The first you've already seen. Simply use `xsl:apply-templates` with a `select` attribute that chooses the particular elements you want to include, like this:

```
<xsl:template match="PERIODIC_TABLE">
  <xsl:apply-templates select="ATOM"/>
</xsl:template>

<xsl:template match="ATOM">
  <xsl:value-of select="."/>
</xsl:template>
```

The `select="."` in the second template tells the formatter to take the value of the matched element, `ATOM` in this example.

The second option is `xsl:for-each`. The `xsl:for-each` element processes each element chosen by its `select` attribute in turn. However, no additional template is required. For example:

```
<xsl:template match="PERIODIC_TABLE">
  <xsl:for-each select="ATOM">
    <xsl:value-of select="."/>
  </xsl:for-each>
</xsl:template>
```

If the `select` attribute is omitted, then all children of the source node (`PERIODIC_TABLE` in this example) are processed.

```
<xsl:template match="PERIODIC_TABLE">
  <xsl:for-each>
    <xsl:value-of select="ATOM"/>
  </xsl:for-each>
</xsl:template>
```

Patterns for Matching Nodes

The `match` attribute of the `xsl:template` element supports a complex syntax that allows you to express exactly which nodes you do and do not want to match. The `select` attribute of `xsl:apply-templates`, `xsl:value-of`, `xsl:for-each`, `xsl:copy-of`, and `xsl:sort` supports an even more powerful superset of this syntax that allows you to express exactly which nodes you do and do not want to select. Various patterns for matching and selecting nodes are discussed below.

Matching the Root Node

In order that the output document be well-formed, the first thing output from an XSL transformation should be the output document's root element. Consequently, XSL style sheets generally start with a rule that applies to the root node. To specify the root node in a rule, you give its `match` attribute the value `" / "`. For example:

```
<xsl:template match="/">
  <html>
    <xsl:apply-templates/>
  </html>
</xsl:template>
```

This rule applies to the root node and only the root node of the input tree. When the root node is read, the tag `<html>` is output, the children of the root node are processed, then the `</html>` tag is output. This rule overrides the default rule for the root node. Listing 14-6 shows a style sheet with a single rule that applies to the root node.

Listing 14-6: An XSL style sheet with one rule for the root node

```
<?xml version="1.0"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/XSL/Transform/1.0">

  <xsl:template match="/">
    <html>
      <head>
        <title>Atomic Number vs. Atomic Weight</title>
      </head>
      <body>
        <table>
          Atom data will go here
```

Continued

Listing 14-6: (continued)

```

        </table>
    </body>

</html>
</xsl:template>

</xsl:stylesheet>

```

Since this style sheet only provides a rule for the root node, and since that rule's template does not specify any further processing of child nodes, only literal output, what you see in the template is all that will be inserted in the result document. In other words, the result of applying the style sheet in Listing 14-6 to Listing 14-1 (or any other well-formed XML document) is this:

```

<html><head><title>Atomic Number vs. Atomic
Weight</title></head><body><table>
    Atom data will go here
</table></body></html>

```

Matching Element Names

As previously mentioned, the most basic pattern contains a single element name which matches all elements with that name. For example, this template matches ATOM elements and marks their ATOMIC_NUMBER children bold:

```

<xsl:template match="ATOM">
    <b><xsl:value-of select="ATOMIC_NUMBER"/><b>
</xsl:template>

```

Listing 14-7 demonstrates a style sheet that expands on Listing 14-6. First, an `xsl:apply-templates` element is included in the template of the rule for the root node. This rule uses a `select` attribute to ensure that only `PERIODIC_TABLE` elements get processed.

Secondly, a rule that only applies to `PERIODIC_TABLE` elements is created using `match="PERIODIC_TABLE"`. This rule sets up the header for the table, then applies templates to form the body of the table from `ATOM` elements.

Finally, the `ATOM` rule specifically selects the `ATOM` element's `NAME`, `ATOMIC_NUMBER`, and `ATOMIC_WEIGHT` child elements with `<xsl:apply-templates select="NAME"/>`, `<xsl:apply-templates select="ATOMIC_NUMBER"/>`, and `<xsl:apply-templates select="ATOMIC_WEIGHT"/>`. These are wrapped up inside HTML's `tr` and `td` elements so that the end result is a table of atomic numbers matched to atomic weights. Figure 14-4 shows the output of applying

the style sheet in Listing 14-7 to the complete periodic table document displayed in Netscape Navigator 4.5.

One thing you may wish to note about this style sheet: the exact order of the `NAME`, `ATOMIC_NUMBER`, and `ATOMIC_WEIGHT` elements in the input document is irrelevant. They appear in the output in the order they were selected; that is, first number, then weight. Conversely, the individual atoms are sorted in alphabetical order as they appear in the input document. Later, you'll see how to use an `xsl:sort` element to change that so you can arrange the atoms in the more conventional atomic number order.

Listing 14-7: Templates applied to specific classes of element with select

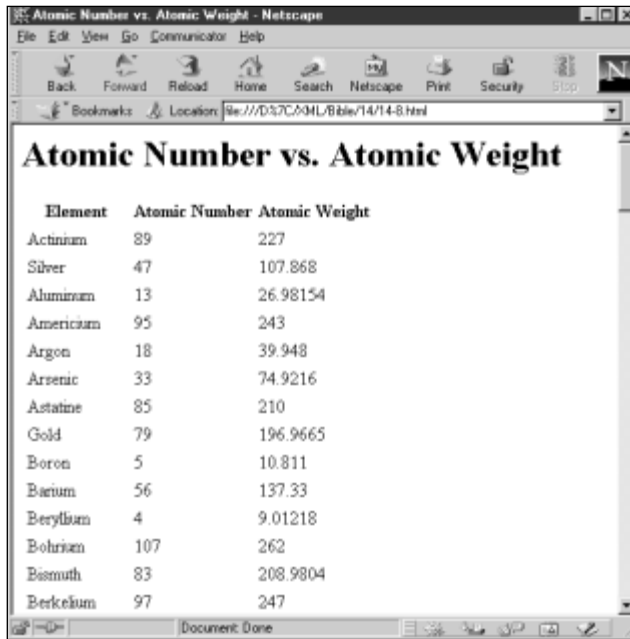
```
<?xml version="1.0"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/XSL/Transform/1.0">

  <xsl:template match="/">
    <html>
      <head>
        <title>Atomic Number vs. Atomic Weight</title>
      </head>
      <body>
        <xsl:apply-templates select="PERIODIC_TABLE"/>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="PERIODIC_TABLE">
    <h1>Atomic Number vs. Atomic Weight</h1>
    <table>
      <th>Element</th>
      <th>Atomic Number</th>
      <th>Atomic Weight</th>
      <xsl:apply-templates select="ATOM"/>
    </table>
  </xsl:template>

  <xsl:template match="ATOM">
    <tr>
      <td><xsl:value-of select="NAME"/></td>
      <td><xsl:value-of select="ATOMIC_NUMBER"/></td>
      <td><xsl:value-of select="ATOMIC_WEIGHT"/></td>
    </tr>
  </xsl:template>

</xsl:stylesheet>
```



The screenshot shows a Netscape Navigator 4.5 window with the title 'Atomic Number vs. Atomic Weight - Netscape'. The address bar shows a local file path. The main content area displays a table with three columns: 'Element', 'Atomic Number', and 'Atomic Weight'. The table lists various elements and their corresponding atomic numbers and weights.

Element	Atomic Number	Atomic Weight
Actinium	89	227
Silver	47	107.868
Aluminum	13	26.98154
Americium	95	243
Argon	18	39.948
Arsenic	33	74.9216
Astatine	85	210
Gold	79	196.9665
Boron	5	10.811
Barium	56	137.33
Beryllium	4	9.01218
Bohrium	107	262
Bismuth	83	208.9804
Berkelium	97	247

Figure 14-4: A table showing atomic number vs. atomic weight in Netscape Navigator 4.5

Matching Children with /

You're not limited to the children of the current node in `match` attributes. You can use the `/` symbol to match specified hierarchies of elements. Used alone, the `/` symbol refers to the root node. However, you can use it between two names to indicate that the second is the child of the first. For example, `ATOM/NAME` refers to `NAME` elements that are children of `ATOM` elements.

In `xsl:template` elements this enables you to match only some of the elements of a given kind. For example, this template rule marks `SYMBOL` elements that are children of `ATOM` elements strong. It does nothing to `SYMBOL` elements that are not direct children of `ATOM` elements.

```
<xsl:template match="ATOM/SYMBOL">
  <strong><xsl:value-of select="."/></strong>
</xsl:template>
```

Caution

Remember that this rule selects `SYMBOL` elements that are children of `ATOM` elements, not `ATOM` elements that have `SYMBOL` children. In other words, the `.` in `<xsl:value-of select="."/>` refers to the `SYMBOL` and not to the `ATOM`.

You can specify deeper matches by stringing patterns together. For example, `PERIODIC_TABLE/ATOM/NAME` selects `NAME` elements whose parent is an `ATOM` element, whose parent is a `PERIODIC_TABLE` element.

You can also use the `*` wild card to substitute for an arbitrary element name in a hierarchy. For example, this template rule applies to all `SYMBOL` elements that are grandchildren of a `PERIODIC_TABLE` element.

```
<xsl:template match="PERIODIC_TABLE/*/SYMBOL">
  <strong><xsl:value-of select="."/></strong>
</xsl:template>
```

Finally, as you saw above, a `/` by itself selects the root node of the document. For example, this rule applies to all `PERIODIC_TABLE` elements that are root elements of the document:

```
<xsl:template match="/PERIODIC_TABLE">
  <html><xsl:apply-templates/></html>
</xsl:template>
```

While `/` refers to the root node, `/*` refers to the root element, whatever it is. For example,

```
<xsl:template match="/*">
  <html>
    <head>
      <title>Atomic Number vs. Atomic Weight</title>
    </head>
    <body>
      <xsl:apply-templates/>
    </body>
  </html>
</xsl:template>
```

Matching Descendants with //

Sometimes, especially with an uneven hierarchy, you may find it easier to bypass intermediate nodes and simply select all the elements of a given type whether they're immediate children, grandchildren, great-grandchildren, or what have you. The double slash, `//`, refers to a descendant element at an arbitrary level. For example, this template rule applies to all `NAME` descendants of `PERIODIC_TABLE`, no matter how deep:

```
<xsl:template match="PERIODIC_TABLE//NAME">
  <i><xsl:value-of select="."/></i>
</xsl:template>
```

The periodic table example is fairly shallow, but this trick becomes more important in deeper hierarchies, especially when an element can contain other elements of its type (for example, an `ATOM` contains an `ATOM`).

The `//` operator at the beginning of a pattern selects any descendant of the root node. For example, this template rule processes all `ATOMIC_NUMBER` elements while completely ignoring their location:

```
<xsl:template match="//ATOMIC_NUMBER">
  <i><xsl:value-of select="."/></i>
</xsl:template>
```

Matching by ID

You may want to apply a particular style to a particular single element without changing all other elements of that type. The simplest way to do that in XSL is to attach a style to the element's ID attribute. This is done with the `id()` selector which contains the ID value in single quotes. For example, this rule makes the element with the ID `e47` bold:

```
<xsl:template match="id('e47')">
  <b><xsl:value-of select="."/></b>
</xsl:template>
```

This assumes of course that the elements you want to select in this fashion have an attribute declared as type `ID` in the source document's DTD. Usually this isn't the case, however. For one thing, many documents do not have DTDs. They're merely well-formed, not valid. And even if they have a DTD, there's no guarantee that any element has an `ID` type attribute. You can use the `xsl:key` element in the style sheet to declare that particular attributes in the input document should be treated as IDs.

Matching Attributes with @

As you already saw back in Chapter 5, the `@` sign matches against attributes and selects nodes according to attribute names. Simply prefix the attribute you want to select with the `@` sign. For example, Listing 14-8 shows a style sheet that outputs a table of atomic number vs. melting point. Not only is the value of the `MELTING_POINT` element written out, but also the value of its `UNITS` attribute. This is selected by `<xsl:value-of select="@UNITS"/>`.

Listing 14-8: An XSL style sheet that selects the `UNITS` attribute with `@`

```
<?xml version="1.0"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/XSL/Transform/1.0">

  <xsl:template match="/PERIODIC_TABLE">
    <html>
      <body>
```

```

        <h1>Atomic Number vs. Melting Point</h1>
        <table>
            <th>Element</th>
            <th>Atomic Number</th>
            <th>Melting Point</th>
            <xsl:apply-templates/>
        </table>
    </body>
</html>
</xsl:template>

<xsl:template match="ATOM">
    <tr>
        <td><xsl:value-of select="NAME"/></td>
        <td><xsl:value-of select="ATOMIC_NUMBER"/></td>
        <td><xsl:apply-templates select="MELTING_POINT"/></td>
    </tr>
</xsl:template>

<xsl:template match="MELTING_POINT">
    <xsl:value-of select="."/>
    <xsl:value-of select="@UNITS"/>
</xsl:template>

</xsl:stylesheet>

```

Recall that the value of an attribute node is simply the string value of the attribute. Once you apply the style sheet in Listing 14-8, `ATOM` elements come out formatted like this:

```

<tr><td>Hydrogen</td><td>1</td><td>13.81Kelvin</td></tr>

<tr><td>Helium</td><td>2</td><td>0.95Kelvin</td></tr>

```

You can combine attributes with elements using the various hierarchy operators. For example, `BOILING_POINT/@UNITS` refers to the `UNITS` attribute of a `BOILING_POINT` element. `ATOM/*/@UNITS` matches any `UNITS` attribute of a child element of an `ATOM` element. This is especially helpful when matching against attributes in template rules. You must remember that what's being matched is the attribute node, not the element that contains it. It's a very common mistake to implicitly confuse the attribute node with the element node that contains it. For example, consider this rule, which attempts to apply templates to all child elements that have `UNITS` attributes:

```

<xsl:template match="ATOM">
    <xsl:apply-templates select="@UNITS"/>
</xsl:template>

```

What it actually does is apply templates to the non-existent `UNITS` attributes of `ATOM` elements.

You can also use the `*` to select all attributes of an element, for example `BOILING_POINT/@*` to select all attributes of `BOILING_POINT` elements.

Matching Comments with `comment()`

Most of the time you should simply ignore comments in XML documents. Making comments an essential part of a document is a very bad idea. Nonetheless, XSL does provide a means to select a comment if you absolutely have to.

To select a comment, use the pattern `comment()`. Although this pattern has function-like parentheses, it never actually takes any arguments. You cannot easily distinguish between different comments. For example, recall that a `DENSITY` element looks like this:

```
<DENSITY UNITS="grams/cubic centimeter"><!-- At 300K -->
  6.51
</DENSITY>
```

This template rule not only outputs the value of the density and the units; it also prints the conditions under which the density is measured:

```
<xsl:template match="DENSITY">
  <xsl:value-of select="."/>
  <xsl:value-of select="@UNITS"/>
  <xsl:apply-templates select="comment()"/>
</xsl:template>
```

The only reason Listing 14-1 uses a comment to specify conditions instead of an attribute or element is precisely for this example. In practice, you should never put important information in comments. The only real reason XSL ever allows you to select comments is so a style sheet can transform from one markup language to another while leaving the comments intact. Any other use indicates a poorly designed original document. The following rule matches all comments, and copies them back out again using the `xsl:comment` element.

```
<xsl:template match="comment()">
  <xsl:comment><xsl:value-of select="."/></xsl:comment>
</xsl:template>
```

Note, however, that the default rules used to apply templates do not apply to comments. Thus, if you want this rule to be activated when a comment is encountered, you'll need to include an `xsl:apply-templates` element that selects comments wherever comments may be found.

You can use the hierarchy operators to select particular comments. For example, this rule matches comments that occur inside `DENSITY` elements:

```
<xsl:template match="DENSITY/comment()">
  <xsl:comment><xsl:value-of select="."/></xsl:comment>
</xsl:template>
```

Matching Processing Instructions with pi()

When it comes to writing structured, intelligible, maintainable XML, processing instructions aren't much better than comments. However, there are some necessary uses for them including attaching style sheets to documents.

The `pi()` function selects processing instructions. The argument to `pi()` is a quoted string giving the name of the processing instruction to select. If you do not include an argument, the first processing instruction child of the current node is matched. However, you can use hierarchy operators. For example, this rule matches the first processing instruction child of the root node (most likely the `xml-styleSheet` processing instruction). The `xsl:pi` element inserts a processing instruction with the specified name and value in the output document.

```
<xsl:template match="/pi()">
  <xsl:pi name="xml-styleSheet">
    type="text/xml" value="auto.xml"
  </xsl:pi>
</xsl:template/>
```

This rule also matches the `xml-styleSheet` processing instruction, but by its name:

```
<xsl:template match="pi('xml-styleSheet')">
  <xsl:pi name="xml-styleSheet">
    <xsl:value-of select="."/>
  </xsl:pi>
</xsl:template/>
```

In fact, one of the main reasons for distinguishing between the root element and the root node is so that processing instructions from the prolog can be read and processed. Although the `xml-styleSheet` processing instruction uses a `name=value` syntax, XSL does not consider these to be attributes because processing instructions are not elements. The value of a processing instruction is simply everything between the whitespace following its name and the closing `?`.

The default rules used to apply templates do not match processing instructions. Thus, if you want this rule to be activated when the `xml-styleSheet` processing instruction is encountered, you'll need to include an `xsl:apply-templates` element that matches it in the appropriate place. For example, this template rule for the root node does apply templates to processing instructions:

```
<xsl:template match="/">
  <xsl:apply-templates select="pi()"/>
  <xsl:apply-templates select="*"/>
</xsl:template>
```

Matching Text Nodes with text()

Text nodes generally get ignored as nodes, though their values are included as part of the value of a selected element. However, the `text()` operator does enable you to specifically select the text child of an element. Despite the parentheses, this operator takes no arguments. For example:

```
<xsl:template match="SYMBOL">
  <xsl:value-of select="text()"/>
</xsl:template>
```

The main reason this operator exists is for the default rules. XSL processors must provide the following default rule whether the author specifies it or not:

```
<xsl:template match="text()">
  <xsl:value-of select="."/>
</xsl:template>
```

This means that whenever a template is applied to a text node, the text of the node is output. If you do not want the default behavior, you can override it. For example, including the following empty template rule in your style sheet will prevent text nodes from being output unless specifically matched by another rule.

```
<xsl:template match="text()">
</xsl:template>
```

Using the Or Operator |

The vertical bar (`|`) allows a template rule to match multiple patterns. If a node matches one pattern or the other, it will activate the template. For example, this template rule matches both `ATOMIC_NUMBER` and `ATOMIC_WEIGHT` elements:

```
<xsl:template match="ATOMIC_NUMBER|ATOMIC_WEIGHT">
  <B><xsl:apply-templates/></B>
</xsl:template>
```

You can include whitespace around the `|` if that makes the code clearer. For example,

```
<xsl:template match="ATOMIC_NUMBER | ATOMIC_WEIGHT">
  <B><xsl:apply-templates/></B>
</xsl:template>
```

You can also use more than two patterns in sequence. For example, this template rule applies to `ATOMIC_NUMBER`, `ATOMIC_WEIGHT`, and `SYMBOL` elements (that is, it matches `ATOMIC_NUMBER`, `ATOMIC_WEIGHT` and `SYMBOL` elements):

```
<xsl:template match="ATOMIC_NUMBER | ATOMIC_WEIGHT | SYMBOL">
  <B><xsl:apply-templates/></B>
</xsl:template>
```

The `/` operator is evaluated before the `|` operator. Thus, the following template rule selects an `ATOMIC_NUMBER` child of an `ATOM`, or an `ATOMIC_WEIGHT` of unspecified parentage, not an `ATOMIC_NUMBER` child of an `ATOM` or an `ATOMIC_WEIGHT` child of an `ATOM`.

```
<xsl:template match="ATOM/ATOMIC_NUMBER|ATOMIC_WEIGHT">
  <B><xsl:apply-templates/></B>
</xsl:template>
```

Testing with []

So far, we've merely tested for the presence of various nodes. However, you can test for more details about the nodes that match a pattern using `[]`. You can perform many different tests including:

- ♦ Whether an element contains a given child, attribute, or other node
- ♦ Whether the value of an attribute is a given string
- ♦ Whether the value of an element matches a string
- ♦ What position a given node occupies in the hierarchy

For example, seaborgium, element 106, has only been created in microscopic quantities. Even its most long-lived isotope has a half-life of only 20 seconds. With such a hard-to-create, short-lived element, it's virtually impossible to measure the density, melting point, or other bulk properties. Consequently, the periodic table document omits the elements describing the bulk properties of seaborgium and similar atoms as a result of unavailable data. If you want to create a table of atomic number vs. melting point, you should omit those elements with unknown melting points. To do this, you can specify a match against `ATOM` elements that have `MELTING_POINT` children like this:

```
<xsl:template match="ATOM[MELTING_POINT]">
  <tr>
    <td><xsl:value-of select="NAME"/></td>
    <td><xsl:value-of select="MELTING_POINT"/></td>
  </tr>
</xsl:template>
```

Note here, that it is the `ATOM` element being matched, not the `MELTING_POINT` element as in the case of `ATOM/MELTING_POINT`.

The test brackets can contain more than simply a child element name. In fact, they can contain any select expression. (Select expressions are a superset of match patterns that will be discussed in the next section.) If the specified element has a child matching that expression, it is considered to match the total pattern. For example, this template rule matches `ATOM` elements with `NAME` or `SYMBOL` children.

```
<xsl:template match="ATOM[NAME | SYMBOL]">
</xsl:template>
```

This template rule uses a `*` to match any element that contains a `NAME` child:

```
<xsl:template match="*[NAME]">
  </xsl:template>
```

This template rule matches `ATOM` elements with a `DENSITY` child that has a `UNITS` attribute:

```
<xsl:template match="ATOM[DENSITY/@UNITS]">
  </xsl:template>
```

To revisit an earlier example, to correctly find all child elements that have `UNITS` attributes, use `*` to find all elements and `[@UNITS]` to winnow those down to the ones with `UNITS` attributes, like this:

```
<xsl:template match="ATOM">
  <xsl:apply-templates select="*[@UNITS]"/>
</xsl:template>
```

One type of pattern testing that proves especially useful is string equality. An equals sign (`=`) can test whether the value of a node identically matches a given string. For example, this template finds the `ATOM` element that contains an `ATOMIC_NUMBER` element whose contents include the string 10 (Neon).

```
<xsl:template match="ATOM[ATOMIC_NUMBER='10']">
  This is Neon!
</xsl:template>
```

Testing against element content may seem extremely tricky because of the need to get the value exactly right, including whitespace. You may find it easier to test against attribute values since those are less likely to contain insignificant whitespace. For example, the style sheet in Listing 14-9 applies templates only to those `ATOM` elements whose `STATE` attribute value is the three letters `GAS`.

Listing 14-9: An XSL style sheet that selects only those `ATOM` elements whose `STATE` attribute has the value `GAS`

```
<?xml version="1.0"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/XSL/Transform/1.0">

  <xsl:template match="PERIODIC_TABLE">
    <html>
      <head><title>Gases</title></head>
      <body>
        <xsl:apply-templates select="ATOM[@STATE='GAS']"/>
      </body>
    </html>
```

```

</xsl:template>

<xsl:template match="ATOM">
  <P><xsl:value-of select="."/></P>
</xsl:template>

</xsl:stylesheet>

```

You can use other XSL expressions (discussed in the next section) for more complex matches. For example, you can select all elements whose names begin with “A” or all elements with an atomic number less than 100.

Expressions for Selecting Nodes

The `select` attribute is used in `xsl:apply-templates`, `xsl:value-of`, `xsl:for-each`, `xsl:copy-of`, and `xsl:sort` to specify exactly which nodes are operated on. The value of this attribute is an *expression*. Expressions are a superset of the match patterns discussed in the last section. That is, all match patterns are select expressions, but not all select expressions are match patterns. Recall that match patterns enable you to match nodes by element name, child elements, descendants, and attributes, as well by making simple tests on these items. Select expressions allow you to select nodes through all these criteria but also by referring to parent elements, sibling elements, and making much more complicated tests. Furthermore, expressions aren’t limited to producing merely a list of nodes, but can also produce booleans, numbers, and strings.

Node Axes

Expressions are not limited to specifying the children and descendants of the current node. XSL provides a number of axes you can use to select from different parts of the tree relative to the current node (generally the node that the template matches). Table 14-2 summarizes these axes and their meanings.

Table 14-2
Expression Axes

<i>Axis</i>	<i>Selects From</i>
<code>from-ancestors()</code>	the parent of the current node, the parent of the parent of the current node, the parent of the parent of the parent of the current node, and so forth back to the root node

Continued

Table 14-2 (continued)

Axis	Selects From
from-ancestors-or-self()	the ancestors of the current node and the current node itself
from-attributes()	the attributes of the current node
from-children()	the immediate children of the current node
from-descendants()	the children of the current node, the children of the children of the current node, and so forth
from-descendants-or-self()	the current node itself and its descendants
from-following()	all nodes that start after the end of the current node
from-following-siblings()	all nodes that start after the end of the current node and have the same parent as the current node
from-parent()	the single parent node of the current node
from-preceding()	all nodes that start before the start of the current node
from-preceding-siblings()	all nodes that start before the start of the current node and have the same parent as the current node
from-self()	the current node

**Caution**

The from-following and from-preceding axes are on the questionable side. They may not be included in the final release of XSL. If they are included, their exact meaning may change.

These axes serve as functions that select from the set of nodes indicated in the second column of Table 14-2. The parentheses contain a select expression to further winnow down this node list. For example, they may contain the name of the element to be selected as in the following template rule:

```
<xsl:template match="ATOM">
  <tr>
    <td>
      <xsl:value-of select="from-children(NAME)"/>
    </td>
    <td>
      <xsl:value-of select="from-children(ATOMIC_NUMBER)"/>
    </td>
    <td>
      <xsl:value-of select="from-children(ATOMIC_WEIGHT)"/>
    </td>
  </tr>
</xsl:template>
```

The template rule matches `ATOM` elements. When an `ATOM` element is matched, a `NAME` element, an `ATOMIC_NUMBER` element, and an `ATOMIC_WEIGHT` element are all selected from the children of that matched `ATOM` element and output as table cells. (If there's one more than one of these desired elements—for example, three `NAME` elements—then only the first one is selected.)

The `from-children()` axis doesn't let you do anything you can't do with element names alone. In fact `select="ATOMIC_WEIGHT"` is just an abbreviated form of `select="from-children(ATOMIC_WEIGHT)"`. However, the other axes are a little more interesting.

Referring to the parent element is illegal in match patterns, but not in select expressions. To refer to the parent, you use the `from-parent()` axis. For example, this rule outputs the value of atoms that have a `BOILING_POINT` child:

```
<xsl:template match="ATOM/BOILING_POINT">
  <P><xsl:value-of select="from-parent(ATOM)"/></P>
</xsl:template>
```

Here the `BOILING_POINT` child element is matched, but the `ATOM` parent element is output.

Some radioactive atoms like polonium have half-lives so short that bulk properties like the boiling point and melting point can't be measured. Therefore, not all `ATOM` elements will necessarily have `BOILING_POINT` child elements. The above rule would allow you to only output those elements that actually have boiling points. Expanding on this example, Listing 14-10 matches the `MELTING_POINT` elements but actually outputs the parent `ATOM` element using `from-parent(ATOM)`.

Listing 14-10: A style sheet that outputs only those elements with known melting points

```
<?xml version="1.0"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/XSL/Transform/1.0">

  <xsl:template match="/">
    <html>
      <body>
        <xsl:apply-templates select="PERIODIC_TABLE"/>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="PERIODIC_TABLE">
    <h1>Elements with known Melting Points</h1>
```

Continued

Listing 14-10: (continued)

```
        <xsl:apply-templates select="//MELTING_POINT"/>
    </xsl:template>

    <xsl:template match="MELTING_POINT">
        <p>
            <xsl:value-of select="from-parent(ATOM)"/>
        </p>
    </xsl:template>

</xsl:stylesheet>
```

Once in awhile, you may need to select the nearest ancestor of an element with a given type. The `from-ancestors()` function does this. For example, this rule inserts the value of the nearest `PERIODIC_TABLE` element that contains the matched `SYMBOL` element.

```
<xsl:template match="SYMBOL">
    <xsl:value-of select="from-ancestors(PERIODIC_TABLE)"/>
</xsl:template>
```

The `from-ancestors-or-self()` function behaves like the `from-ancestors()` function except that if the current node matches the type of the argument, then it will be returned instead of the actual ancestor. For example, this rule matches all elements. If the matched element is a `PERIODIC_TABLE`, then that very `PERIODIC_TABLE` is selected in `xsl:value-of`.

```
<xsl:template match="*">
    <xsl:value-of select="from-ancestors-or-self(PERIODIC_TABLE)"/>
</xsl:template>
```

Node Types

As well as the name of a node and the wild card, the arguments to a `from-axis()` function may be one of these four node-type functions:

- ♦ `comment()`
- ♦ `text()`
- ♦ `pi()`
- ♦ `node()`

The `comment()` node type selects a comment node. The `text()` node type selects a text node. The `pi()` node type selects a processing instruction node, and the `node()` node type selects any type of node. (The `*` wild card only selects element

nodes.) The `pi()` node type can also contain an optional argument specifying the name of the processing instruction to select.

For example, this rule wraps the value of a matched `ATOM` element in a `P` element using `from-self()` with the `node()` node type:

```
<xsl:template match="ATOM">
  <P><xsl:value-of select="from-self(node())"/></P>
</xsl:template>
```

Here, selecting `from-self(node())` is not the same as selecting `ATOM`. The next rule tries to take the value of the `ATOM` child of an `ATOM` element. This is not the value of the matched `ATOM` element, but rather the value of a different `ATOM` element that's a child of the matched `ATOM` element:

```
<xsl:template match="ATOM">
  <P><xsl:value-of select="ATOM"/></P>
</xsl:template>
```

Hierarchy Operators

You can use the `/` and `//` operators to string select expressions together. For example, Listing 14-11 prints a table of element names, atomic numbers, and melting points for only those elements that have melting points. It does this by selecting the parent of the `MELTING_POINT` element, then finding that parent's `NAME` and `ATOMIC_NUMBER` children with `select="from-parent(*)/from-children(NAME)".`

Listing 14-11: A table of melting point versus atomic number

```
<?xml version="1.0"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/XSL/Transform/1.0">

  <xsl:template match="/PERIODIC_TABLE">
    <html>
      <body>
        <h1>Atomic Number vs. Melting Point</h1>
        <table>
          <th>Element</th>
          <th>Atomic Number</th>
          <th>Melting Point</th>
          <xsl:apply-templates select="from-children(ATOM)"/>
        </table>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="ATOM">
```

Continued

Listing 14-11 (continued)

```

        <xsl:apply-templates
          select="from-children(MELTING_POINT)"/>
      </xsl:template>

      <xsl:template match="MELTING_POINT">
        <tr>
          <td>
            <xsl:value-of
              select="from-parent(*)/from-children(NAME)"/>
          </td>
          <td>
            <xsl:value-of
              select="from-parent(*)/from-children(ATOMIC_NUMBER)"/>
          </td>
          <td>
            <xsl:value-of select="from-self(*)"/>
            <xsl:value-of select="from-attributes(UNITS)"/>
          </td>
        </tr>
      </xsl:template>

    </xsl:stylesheet>

```

This is not the only way to solve the problem. Another possibility would be to use the `from-preceding-siblings()` and `from-following-siblings()` axes or both if the relative location (preceding or following) is uncertain. The necessary template rule for the `MELTING_POINT` element would look like this:

```

    <xsl:template match="MELTING_POINT">
      <tr>
        <td>
          <xsl:value-of
            select="from-preceding-siblings(NAME)
                  | from-following-siblings(NAME)"/>
        </td>
        <td>
          <xsl:value-of
            select="from-preceding-siblings(ATOMIC_NUMBER)
                  | from-following-siblings(ATOMIC_NUMBER)"/>
        </td>
        <td>
          <xsl:value-of select="from-self(*)"/>
          <xsl:value-of select="from-attributes(UNITS)"/>
        </td>
      </tr>
    </xsl:template>

```

Abbreviated Syntax

The various `from-axis()` functions in Table 14-2 are a bit too wordy for comfortable typing. XSL also defines an abbreviated syntax that can substitute for the most common of these axes and is more used in practice. Table 14-3 shows the full and abbreviated equivalents.

Table 14-3
Abbreviated Syntax for Select Expressions

<i>Abbreviation</i>	<i>Full</i>
<code>.</code>	<code>from-self(node())</code>
<code>..</code>	<code>from-parent(node())</code>
<code><i>name</i></code>	<code>from-children(<i>name</i>)</code>
<code>@<i>name</i></code>	<code>from-attributes(<i>name</i>)</code>
<code>//</code>	<code>/from-descendants-or-self(node())</code>

Listing 14-12 demonstrates by rewriting Listing 14-11 using the abbreviated syntax. The output produced by the two style sheets is exactly the same, however.

Listing 14-12: A table of melting point versus atomic number using the abbreviated syntax

```
<?xml version="1.0"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/XSL/Transform/1.0">

  <xsl:template match="/PERIODIC_TABLE">
    <html>
      <body>
        <h1>Atomic Number vs. Melting Point</h1>
        <table>
          <th>Element</th>
          <th>Atomic Number</th>
          <th>Melting Point</th>
          <xsl:apply-templates select="ATOM"/>
        </table>
      </body>
    </html>
  </xsl:template>
```

Continued

Listing 14-12 (continued)

```
<xsl:template match="ATOM">
  <xsl:apply-templates
    select="MELTING_POINT"/>
</xsl:template>

<xsl:template match="MELTING_POINT">
  <tr>
    <td>
      <xsl:value-of
        select="../NAME"/>
    </td>
    <td>
      <xsl:value-of
        select="../ATOMIC_NUMBER"/>
    </td>
    <td>
      <xsl:value-of select="."/>
      <xsl:value-of select="@UNITS"/>
    </td>
  </tr>
</xsl:template>

</xsl:stylesheet>
```

Match patterns can only use the abbreviated syntax (and not all of that). The full syntax using the *from-axis()* functions of Table 14-2 is restricted to select expressions.

Expression Types

Every expression evaluates to a single value. For example, the expression $3 + 2$ evaluates to the value 5. The expression used above all evaluated to node sets. However, there are five types of expressions in XSL. These are:

- ♦ Node-sets
- ♦ Booleans
- ♦ Numbers
- ♦ Strings
- ♦ Result tree fragments

Node Sets

A node set is a list of nodes from the input document. The *from-axis()* functions in Table 14-2 all return a node set containing the node they match. Which nodes are in the node set one of these functions returns depends on the current node (also known as the context node), the argument to the function, and of course, which function it is.

Tip

Programmers accustomed to object-oriented languages like Java and C++ can think of the current node as the object which invokes the function; that is, in `a.doSomething(b, c)` the current node is `a`. However, in XSL the current node is always implicit; that is, it's written more like `doSomething(b, c)` as might be done in the file where `a`'s class is defined.

For example, the expression `select="from-children(ATOM)"` returns a node set that contains both `ATOM` elements in that document when the current node is the `PERIODIC_TABLE` element of Example 14-1. The expression `select="from-children(ATOM)"/from-children("NAME")` returns a node set containing the two element nodes `<NAME>Hydrogen</NAME>` and `<NAME>Helium</NAME>` when the context node is the `PERIODIC_TABLE` element of Example 14-1.

The context node is a member of the *context node list*. The context node list is that group of elements that all match the same rule at the same time, generally as a result of one `xsl:apply-templates` or `xsl:for-each` call. For instance, when Listing 14-12 is applied to Listing 14-1, the `ATOM` template is invoked twice, first for the hydrogen atom, second for the helium atom. The first time it's invoked, the context node is the hydrogen `ATOM` element. The second time it's invoked, the context node is the helium `ATOM` element. However, both times the context node list is the set containing both the helium and hydrogen `ATOM` elements.

Table 14-4 lists a number of functions that operate on node-sets, either as arguments or as the context node.

Table 14-4
Functions That Operate on Node Sets

<i>Function</i>	<i>Return Type</i>	<i>Returns</i>
<code>position()</code>	number	The position of the context node in the context node list. The first node in the list has position 1.
<code>last()</code>	number	The number of nodes in the context node set.
<code>count(<i>node-set</i>)</code>	number	The number of nodes in <i>node-set</i> .

Continued

Table 14-4 (continued)

Function	Return Type	Returns
<code>id(string)</code>	node set	A node set containing the single element anywhere in the same document whose ID is <i>string</i> ; or the empty set if no element has the specified ID.
<code>idref(node-set)</code>	node set	A node set containing all of the elements in the document whose ID is one of the tokens (separated by whitespace) in the values of the nodes in the argument <i>node-set</i> .
<code>key(string name, string value)</code>	node set	A node set containing all nodes in this document that have a key with the specified value. Keys are set with the top-level <code>xml:key</code> element.
<code>keyref(string name, node set values)</code>	node set	A node set containing all nodes in this document that have a key whose value is the same as the value of one of the nodes in the second argument.
<code>doc(string URI)</code>	node set	A node set in the document or portion referred to by the URI; the nodes are chosen from the named anchor or XPointer used by the URI. If there is no named anchor or Xpointer, then the root element of the named document is in the node set. Relative URIs are relative to the current node in the input document.
<code>docref(node set)</code>	node set	A node set containing all the nodes (in one or more documents) referred to by the URIs that are the value of the <i>node set</i> argument.
<code>local-part (node set)</code>	string	The local part (everything after the namespace prefix) of the first node in the <i>node set</i> argument; can be used without any arguments to get the local part of the context node.
<code>namespace(node set)</code>	string	The URI of the namespace of the first node in the node set; can be used without any arguments to get the URI of the namespace of the context node; returns an empty string if the node is in the default namespace.
<code>qname(node set)</code>	string	The qualified name (both prefix and local part) of the first node in the <i>node set</i> argument; can be used without any argument to get the qualified name of the context node.
<code>generate-id (node set)</code>	string	A unique identifier for the first node in the argument <i>node set</i> ; can be used without any argument to generate an ID for the context node.

 Cross-Reference

Chapter 18, *Namespaces*, discusses namespace URIs, prefixes, and local parts.

 Caution

The `doc()` and `docref()` functions are a little hazy, especially if their URIs refer to only fragments of a node or data that is not well-formed XML. The details remain to be cleaned up in a future version of the XSL specification.

If an argument of the wrong type is passed to one of these functions, then XSL will attempt to convert that argument to the correct type; for instance, by converting the number 12 to the string “12”. However, no arguments may be converted to node sets.

The `position()` function can be used to count elements. Listing 14-13 is a style sheet that prefixes the name of each atom’s name with its position in the document using `<xsl:value-of select="position()"/>`.

Listing 14-13: A style sheet that numbers the atoms in the order they appear in the document

```
<?xml version="1.0"?>
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/XSL/Transform/1.0">

  <xsl:template match="/PERIODIC_TABLE">
    <HTML>
      <HEAD><TITLE>The Elements</TITLE></HEAD>
      <BODY>
        <xsl:apply-templates select="ATOM"/>
      </BODY>
    </HTML>
  </xsl:template>

  <xsl:template match="ATOM">
    <P>
      <xsl:value-of select="position()"/>.
      <xsl:value-of select="NAME"/>
    </P>
  </xsl:template>

</xsl:stylesheet>
```

When this style sheet is applied to Listing 14-1, the output is this:

```
<HTML><HEAD><TITLE>The Elements</TITLE></HEAD><BODY><P>1.
  Hydrogen</P><P>2.
  Helium</P></BODY></HTML>
```

Booleans

A Boolean has one of two values, true or false. XSL allows any kind of data to be transformed into a Boolean. This is often done implicitly when a string or a number or a node set is used where a Boolean is expected, as in the `test` attribute of an `xsl:if` element. These conversions can also be performed by the `boolean()` function which converts an argument of any type (or the context node if no argument is provided) to a Boolean according to these rules:

- ♦ a number is false if it's zero or NaN (a special symbol meaning “Not a Number”, used for the result of dividing by zero and similar illegal operations), true otherwise
- ♦ an empty node set is false; all other node sets are true
- ♦ an empty result fragment is false; all other result fragments are true
- ♦ a zero length string is false; all other strings are true

Booleans are also produced as the result of expressions involving these operators:

=	equality
<	less-than (really < ;)
>	greater-than
<=	less-than or equal to (really < ;=)
>=	greater-than or equal to



Caution

The `<` sign is illegal in attribute values. Consequently, it must be replaced by `<` even when used as the less-than operator.

These operators are most commonly used in predicate tests to determine whether a rule is invoked. A select expression can contain not only a pattern that selects certain nodes, but also a predicate that further filters the list of nodes selected. For example, `from-children(ATOM)` selects all the `ATOM` children of the current node. However, `from-children(ATOM[position()=1])` selects only the first `ATOM` child of the current node. `[position()=1]` is a predicate on the node test `ATOM` that returns a Boolean result, true if the position of the current node is equal to one, false otherwise. Each node test can have any number of predicates. However, more than one is unusual.

For example, this template rule applies to the first `ATOM` element in the periodic table, but not to subsequent ones, by testing whether or not the position of the element equals 1.

```
<xsl:template match="PERIODIC_TABLE/ATOM[position()=1]">
  <xsl:value-of select="."/>
</xsl:template>
```

This template rule applies to all `ATOM` elements that are not the first child element of the `PERIODIC_TABLE` by testing whether the position is greater than 1:

```
<xsl:template match="PERIODIC_TABLE/ATOM[position()>1]">
  <xsl:value-of select="."/>
</xsl:template>
```

The keywords `and` and `or` logically combine two Boolean expressions according to the normal rules of logic. For example, suppose you want to apply a template to an `ATOMIC_NUMBER` element that is both the first and last child of its parent element; that is, it is the only element of its parent. This template rule uses `and` to accomplish that:

```
<xsl:template
  match="ATOMIC_NUMBER[position()=1 and position()=last()]">
  <xsl:value-of select="."/>
</xsl:template>
```

This template rule applies to both the first and last `ATOM` elements in their parent by matching when the position is 1 or when the position is last:

```
<xsl:template match="ATOM[position()=1 or position()=last()]">
  <xsl:value-of select="."/>
</xsl:template>
```

This is logical `or`, so it will also match if both conditions are true. That is, it will match an `ATOM` that is both the first and last child of its parent

There is no `not` keyword in XSL, but there is a `not()` function. You can reverse the sense of an operation by enclosing it in `not()`. For example, this template rule selects all `ATOM` elements that are not the first child of their parents:

```
<xsl:template match="ATOM[not(position()=1)]">
  <xsl:value-of select="."/>
</xsl:template>
```

This template rule selects all `ATOM` elements that are neither the first nor last `ATOM` child of their parent:

```
<xsl:template match =
  "ATOM[not(position()=1 or position()=last())]">
  <xsl:value-of select="."/>
</xsl:template>
```

There is no exclusive `or` operator. However, one can be formed by judicious use of `not()` and, `and` or. For example, this rule selects those `ATOM` elements that are either the first or last child, but not both

```
<xsl:template
  match="ATOM[(position()=1 or position()=last())
    and not(position()=1 and position()=last())]">
```

```
<xsl:value-of select="."/>
</xsl:template>
```

There are three remaining functions that return Booleans:

- ♦ `true()` always returns true
- ♦ `false()` always returns false
- ♦ `lang(code)` returns true if the current node has the same language (as given by the `xml:lang` attribute) as the *code* argument.

Numbers

XSL numbers are 64-bit IEEE floating-point doubles. Even numbers like 42 or -7000 that look like integers are stored as doubles. Non-number values like strings and Booleans are converted to numbers as necessary, or by the `number()` function using these rules:

- ♦ Booleans are 1 if true; 0 if false.
- ♦ A string is trimmed of leading and trailing whitespace; then converted to a number in the fashion you would expect; for example, the string “12” is converted to the number 12. If the string cannot be interpreted as a number, then it is converted to 0.
- ♦ Node sets and result fragments are converted to strings; then the string is converted to a number.

For example, this rule only outputs the non-naturally occurring trans-uranium elements; those with atomic numbers greater than 92, which is the atomic number of uranium. The node set produced by `ATOMIC_NUMBER` is implicitly converted to the string value of the current `ATOMIC_NUMBER` node. This string is then converted into a number.

```
<xsl:template match="/PERIODIC_TABLE">
  <HTML>
    <HEAD><TITLE>The TransUranium Elements</TITLE></HEAD>
    <BODY>
      <xsl:apply-templates select="ATOM[ATOMIC_NUMBER>92]" />
    </BODY>
  </HTML>
</xsl:template>
```

XSL provides the standard four arithmetic operators:

- ♦ `+` for addition
- ♦ `-` for subtraction
- ♦ `*` for multiplication
- ♦ `div` for division (the more common `/` is already used for other purposes in XSL)

For example, `<xsl:value-of select="2+2"/>` inserts the string “4” into the output document. These operations are more commonly used as part of a test. For example, this rule selects those elements whose atomic weight is more than twice the atomic number:

```
<xsl:template match="/PERIODIC_TABLE">
  <HTML>
    <BODY>
      <H1>High Atomic Weight to Atomic Number Ratios</H1>
      <xsl:apply-templates
        select="ATOM[ATOMIC_WEIGHT > 2 * ATOMIC_NUMBER]"/>
    </BODY>
  </HTML>
</xsl:template>
```

This template actually prints the ratio of atomic weight to atomic number:

```
<xsl:template match="ATOM">
  <p>
    <xsl:value-of select="NAME"/>
    <xsl:value-of select="ATOMIC_WEIGHT div ATOMIC_NUMBER"/>
  </p>
</xsl:template>
```

XSL also provides two less-familiar binary operators:

- ♦ `mod` for taking the remainder of two numbers
- ♦ `quo` for dividing two numbers, then truncating the fractional part to produce an integer

XSL also includes four functions that operate on numbers:

`floor()` returns the greatest integer smaller than the number
`ceiling()` returns the smallest integer greater than the number
`round()` rounds the number to the nearest integer
`sum()` returns the sum of its arguments

For example, this template rule estimates the number of neutrons in an atom by subtracting the atomic number (the number of protons) from the atomic weight (the weighted average over the natural distribution of isotopes of the of number of neutrons plus the number of protons) and rounding to the nearest integer:

```
<xsl:template match="ATOM">
  <p>
    <xsl:value-of select="NAME"/>
    <xsl:value-of
      select="round(ATOMIC_WEIGHT - ATOMIC_NUMBER)"/>
  </p>
</xsl:template>
```

This rule calculates the average atomic weight of all the atoms in the table by adding all the atomic weights, and then dividing by the number of atoms:

```
<xsl:template match="/PERIODIC_TABLE">
  <HTML>
    <BODY>
      <H1>Average Atomic Weight</H1>
      <xsl:value-of
        select="sum(from-descendants(ATOMIC_WEIGHT))
              div count(from-descendants(ATOMIC_WEIGHT))"/>
    </BODY>
  </HTML>
</xsl:template>
```

Strings

A string is a sequence of Unicode characters. Other data types can be converted to strings using the `string()` function according to these rules:

- ♦ Node sets are converted by concatenating the values of the nodes in the set. The values of the nodes in the set are calculated as by the `xsl:value-of` element according to the rules given in Table 14-1.
- ♦ Result tree fragments are converted by acting as if they’re contained in a single element, and then taking the value of that imaginary element. Again, the value of this element is calculated as by the `xsl:value-of` element according to the rules given in Table 14-1. That is, all the result tree fragment’s text (but not markup) is concatenated.
- ♦ A number is converted to a European-style number string like “-12” or “3.1415292”.
- ♦ Boolean false is converted to the English word “false”. Boolean true is converted to the English word “true”.

Besides `string()`, XSL contains seven functions that operate on strings. These are summarized in Table 14-5.

Table 14-5 Functions That Operate on Strings		
Function	Return Type	Returns
<code>starts-with(<i>main_string</i>, <i>prefix_string</i>)</code>	Boolean	true if <i>main_string</i> starts with <i>prefix_string</i> , false otherwise.
<code>contains(<i>containing_string</i>, <i>contained_string</i>)</code>	Boolean	true if the <i>contained_string</i> is part of the <i>containing_string</i> ; false otherwise

Function	Return Type	Returns
<code>substring-before(<i>string</i>, <i>marker-string</i>)</code>	string	the part of the <i>string</i> from the beginning of the string up to (but not including) the first occurrence of <i>marker-string</i>
<code>substring-after(<i>string</i>, <i>marker-string</i>)</code>	string	the part of the <i>string</i> from the end of the first occurrence of <i>marker-string</i> to the end of <i>string</i>
<code>normalize(<i>string</i>)</code>	string	the <i>string</i> after leading and trailing whitespace is stripped and runs of whitespace are replaced with a single space; if the argument is omitted the string value of the context node is normalized
<code>translate(<i>string</i>, <i>replaced_</i> <i>text</i>, <i>replacement_text</i>)</code>	string	returns <i>string</i> with occurrences of characters in <i>replaced_text</i> replaced by the corresponding characters from <i>replacement_text</i>
<code>concat(<i>string1</i>, <i>string2</i>, ...)</code>	string	returns the concatenation of as many strings as are passed as arguments in the order they were passed
<code>format-number(<i>number</i>, <i>format-string</i>, <i>locale-string</i>)</code>	string	returns the string form of <i>number</i> formatted according to the specified <i>format-string</i> in the locale specified by <i>locale-string</i> as if by Java 1.1's <code>java.text.DecimalFormat</code> class (see http://java.sun.com/products/jdk/1.1/docs/api/java.text.DecimalFormat.html)

Result Tree Fragments

A result tree fragment is a portion of an XML document that is not a complete node or set of nodes. For instance, using the `doc()` function with a URI that points into the middle of an element might produce a result tree fragment. Result functions may also be returned by some extension functions (functions unique to a particular XSL implementation or installation).

Since result tree fragments aren't well-formed XML, you can't do much with them. In fact, the only allowed operations are to convert them to a string or a Boolean using `string()` and `boolean()` respectively.

The Default Template Rules

Having to carefully map the hierarchy of an XML document in an XSL style sheet may be inconvenient. This is especially true if the document does not follow a stable, predictable order like the periodic table but rather throws elements together willy-nilly like many Web pages. In those cases, you should have general rules that can find an element and apply templates to it regardless of where it appears in the source document.

To make this process easier, XSL defines two default template rules implicitly included in all style sheets. The first default rule recursively descends the element tree, applying templates to the children of all elements. This guarantees that all template rules that apply to elements will be instantiated. The second default rule applies to text nodes, copying their value onto the output stream. Together these two rules mean that even a blank XSL style sheet with no elements will still produce the raw character data of the input XML document as output.

The Default Rule for Elements

The first default rule applies to element nodes of any type or the root node:

```
<xsl:template match="*/">
  <xsl:apply-templates/>
</xsl:template>
```

`*/` is XSL shorthand for “any element node or the root node”. The purpose of this rule is to ensure that all elements are recursively processed even if they aren’t reached by following the explicit rules. That is, unless another rule overrides this one (especially for the root element) all element nodes will be processed.

However, once an explicit rule for any parent of an element is present, this rule will not be activated for the child elements unless the template rule for the parent has an `xsl:apply-templates` child. For instance, you can stop all processing by matching the root element and neither applying templates nor using `xsl:for-each` to process the children like this:

```
<xsl:template match="/">
</xsl:template>
```

The Default Rule for Text Nodes

Exceptionally observant readers may have noted several of the examples seem to have output the contents of some elements without actually taking the value of the element they were outputting! These contents were provided by XSL’s default rule for text nodes that occur as element content. This rule is:

```
<xsl:template match="text()">
  <xsl:value-of select="."/>
</xsl:template>
```

This rule matches all text nodes (`match="text()"`) and outputs the value of the text node (`<xsl:value-of select="."/>`). In other words, it copies the text from the input to the output.

This rule ensures that at the very least an element's text is output, even if no rule specifically matches it. Another rule can override this one for specific elements where you want either more or less than the text content of an element.

Implication of the Two Default Rules

Together, the two default rules imply that applying an empty style sheet with only an `xsl:stylesheet` element but no children (such as Listing 14-14) to an XML document will copy all the `#PCDATA` out of the elements in the input to the output. However, this method produces no markup. These are, however, extremely low priority rules. Consequently, any other matches take precedence over these two.

Listing 14-14: An empty XML style sheet

```
<?xml version="1.0"?>
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/XSL/Transform/1.0">

</xsl:stylesheet>
```

Caution

One of the most common sources of confusion about XSL in Internet Explorer 5.0 is that it does not provide either of these default rules. You have to make sure that you explicitly match any node whose contents (including descendants) you want to output.

Deciding What Output to Include

It's often necessary to defer decisions about what markup to emit until the input document has been read. For instance, you may want to change the contents of a `FILENAME` element into the `HREF` attribute of an `A` element, or replace one element type in the input with several different element types in the output depending on the value of one of its attributes. This is accomplished by using of `xsl:element`, `xsl:attribute`, `xsl:pi`, `xsl:comment`, and `xsl:text` elements. XSL instructions

are used in the contents of these elements and attribute value templates are used in the attribute values of these elements to vary their output.

Using Attribute Value Templates

Attribute value templates copy data from element content in the input to attribute values in the style sheet. From there, it can be written to the output. For example, suppose you want to convert the periodic table into empty `ATOM` elements with this attribute-based form:

```
<ATOM NAME="Vanadium"
  ATOMIC_WEIGHT="50.9415"
  ATOMIC_NUMBER="23"
  OXIDATION_STATES="5, 4, 3, 2"
  BOILING_POINT="3650K"
  MELTING_POINT="2163K"
  SYMBOL="V"
  DENSITY="6.11 grams/cubic centimeter"
/>
```

To do this, you'll need to extract the contents of elements in the input document and place those in attribute values in the output document. The first thing you're likely to attempt is something like this:

```
<xsl:template match="ATOM">
  <ATOM NAME="<xsl:value-of select='NAME'/>"
    ATOMIC_WEIGHT="<xsl:value-of select='ATOMIC_WEIGHT'/>"
    ATOMIC_NUMBER="<xsl:value-of select='ATOMIC_NUMBER'/>"
  />
</xsl:template>
```

But this is malformed XML. You can't use the `<` character inside an attribute value. Furthermore, it's extremely difficult to write software that can parse this in its most general case.

Instead, inside attribute values, data enclosed in curly braces `{ }` takes the place of the `xsl:value-of` element. The correct way to write the above is like this:

```
<xsl:template match="ATOM">
  <ATOM NAME="{NAME}"/>
    ATOMIC_WEIGHT="{ATOMIC_WEIGHT}"/>
    ATOMIC_NUMBER="{ATOMIC_NUMBER}"/>
  />
</xsl:template>
```

In the output, `{NAME}` is replaced by the value of the `NAME` child element of the current node. `{ATOMIC_WEIGHT}` is replaced by the value of the `ATOMIC_WEIGHT` child element of the current node. `{ATOMIC_NUMBER}` is replaced by the value of the `ATOMIC_NUMBER` child element, and so on.

Attribute value templates can have more complicated patterns than merely an element name. In fact, you can use any of the string expressions discussed earlier in an attribute value template. For example, this template rule selects `DENSITY` elements in the form used in Listing 14-1.

```
<xsl:template match="DENSITY">
  <BULK_PROPERTY
    NAME="DENSITY"
    ATOM="{../NAME}"
    VALUE="{.}"
    UNITS="@UNITS}"
  />
</xsl:template>
```

It converts them into `BULK_PROPERTY` elements that look like this:

```
<BULK_PROPERTY NAME="DENSITY" ATOM="Helium" VALUE="
  0.1785
" UNITS="grams/cubic centimeter"/>
```

Attribute values are not limited to a single attribute value template. You can combine an attribute value template with literal data or with other attribute value templates. For example, this template rule matches `ATOM` elements and replaces them with their name formatted as a link to a file in the format `H.html`, `He.html`, and so on. The file name is derived from the attribute value template `{SYMBOL}` while the literal data provides the period and extension.

```
<xsl:template match="ATOM">
  <A HREF="{SYMBOL}.html">
    <xsl:value-of select="NAME"/>
  </A>
</xsl:template>
```

More than one attribute value template can be included in an attribute value. For example, this template rule includes the density units as part of the `VALUE` attribute rather than making them a separate attribute:

```
<xsl:template match="DENSITY">
  <BULK_PROPERTY
    NAME="DENSITY"
    ATOM="{../NAME}"
    VALUE="{.} {@UNITS}"
  />
</xsl:template>
```

You can use attribute value templates in the value of most attributes in an XSL style sheet. This is particularly important in `xsl:element`, `xsl:attribute`, and `xsl:pi` elements where attribute value templates allow the designer to defer the decision about exactly what element, attribute, or processing instruction appears in the

output until the input document is read. You cannot use attribute value templates as the value of a `select` or `match` attribute, an `xmlns` attribute, an attribute that provides the name of another XSL instruction element, or an attribute of a top-level element (one that's an immediate child of `xsl:stylesheet`).



Chapter 18, *Namespaces*, discusses `xmlns` attributes.

Inserting Elements into the Output with `xsl:element`

Elements usually get inserted into the output document simply by using the literal elements themselves. For instance, to insert a `P` element you merely type `<P>` and `</P>` at the appropriate points in the style sheet. However, occasionally you need to use details from the input to determine which element to place in the output. This might happen, for example, when making a transformation from a source vocabulary that uses attributes for information to an output vocabulary that uses elements for the same information.

The `xsl:element` element inserts an element into the output document. The name of the element is given by an attribute value template in the `name` attribute of `xsl:element`. The contents of the element derive from the contents of the `xsl:element` element which may include `xsl:attribute`, `xsl:pi`, and `xsl:comment` instructions (all discussed below) to insert these items.

For example, suppose you want to replace the `ATOM` elements with `GAS`, `LIQUID`, and `SOLID` elements, depending on the value of the `STATESTATE` attribute. Using `xsl:element`, a single rule can do this by converting the value of the `STATESTATE` attribute to an element name. This is how it works:

```
<xsl:template match="ATOM">
  <xsl:element name="{@STATESTATE}">
    <NAME><xsl:value-of select="NAME"/></NAME>
    <!-- rules for other children -->
  </xsl:element>
</xsl:template>
```

By using more complicated attribute value templates, you can perform most calculations you might need.

Inserting Attributes into the Output with `xsl:attribute`

You can include attributes in the output document simply by using the literal attributes themselves. For instance, to insert a `DIV` element with an `ALIGN` attribute bearing the value `CENTER`, you merely type `<DIV ALIGN="CENTER">` and `</DIV>` at the appropriate points in the style sheet. However, you frequently have to rely on

data you read from the input to determine an attribute value and sometimes even to determine the attribute name.

For example, suppose you want a style sheet that selects atom names and formats them as links to files named H.html, He.html, Li.html, and so forth like this:

```
<LI><A HREF="H.html">Hydrogen</A></LI>
<LI><A HREF="He.html">Helium</A></LI>
<LI><A HREF="Li.html">Lithium</A></LI>
```

Each different element in the input will have a different value for the HREF attribute. The `xsl:attribute` element calculates an attribute name and value and inserts it into the output. Each `xsl:attribute` element is a child of either an `xsl:element` element or a literal element. The attribute calculated by `xsl:attribute` will be attached to the element calculated by its parent in the output. The name of the attribute is specified by the name attribute of the `xsl:attribute` element. The value of the attribute is given by the contents of the `xsl:attribute` element. For example, this template rule produces the output shown above:

```
<xsl:template match="ATOM">
  <LI><A>
    <xsl:attribute name="HREF">
      <xsl:value-of select="SYMBOL"/>.html
    </xsl:attribute>
    <xsl:value-of select="NAME"/>
  </A></LI>
</xsl:template>
```

All `xsl:attribute` elements must come before any other content of their parent element. You can't add an attribute to an element after you've already started writing out its contents. For example, this template is illegal:

```
<xsl:template match="ATOM">
  <LI><A>
    <xsl:value-of select="NAME"/>
    <xsl:attribute name="HREF">
      <xsl:value-of select="SYMBOL"/>.html
    </xsl:attribute>
  </A></LI>
</xsl:template>
```

Defining Attribute Sets

You often need to apply the same group of attributes to many different elements, of either the same or different classes. For instance, you might want to apply a style attribute to each cell in an HTML table. To make this simpler, you can define one or more attributes as members of an attribute set at the top level of the style sheet with `xsl:attribute-set`, then include that attribute set in an element with `xsl:use`.

For example, this `xsl:attribute-set` element defines an element named `cellstyle` with a `font-family` attribute of New York, Times New Roman, Times, serif and a `font-size` attribute of 12pt.

```
<xsl:attribute-set name="cellstyle">
  <xsl:attribute name="font-family">
    New York, Times New Roman, Times, serif
  </xsl:attribute>
  <xsl:attribute name="font-size">12pt</xsl:attribute>
</xsl:attribute-set>
```

This template rule then applies those attributes to `td` elements in the output. As with `xsl:attribute`, the `xsl:use` element that inserts the attribute set must come before any content that's to be added as a child of `td`.

```
<xsl:template match="ATOM">
  <tr>
    <td>
      <xsl:use attribute-set="cellstyle"/>
      <xsl:value-of select="NAME"/>
    </td>
    <td>
      <xsl:use attribute-set="cellstyle"/>
      <xsl:value-of select="ATOMIC_NUMBER"/>
    </td>
  </tr>
</xsl:template>
```

If an element uses more than one attribute set, then all attributes from all the sets are applied to the element. If more than one attribute set defines the same attribute with different values, then the one from the more important set is used. A style sheet in which multiple attribute sets of the same importance define the same attribute is in error.

Generating Processing Instructions with `xsl:pi`

The `xsl:pi` element places a processing instruction in the output document. The target of the processing instruction is specified by a required `name` attribute. The contents of the `xsl:pi` element become the contents of the processing instruction. For example, this rule replaces `PROGRAM` elements with a `gcc` processing instruction:

```
<xsl:template select="PROGRAM">
  <xsl:pi name="gcc"> -04</xsl:pi>
</xsl:template>
```

`PROGRAM` elements in the input are replaced by this processing instruction in the output:

```
<?gcc -04?>
```

The contents of the `xsl:pi` element can include `xsl:value-of` elements and `xsl:apply-templates` elements provided the result of these instructions is pure text. For example,

```
<xsl:template select="PROGRAM">
  <xsl:pi name="gcc">-04 <xsl:value-of select="NAME"/></xsl:pi>
</xsl:template>
```

One common use for `xsl:pi` is to insert the XML declaration when generating XML from XML (even though the XML declaration is technically not a processing instruction). For example:

```
<xsl:pi name="xml">version="1.0" standalone="yes"</xsl:pi>
```

The `xsl:pi` element may not contain `xsl:element` and other instructions that produce elements and attributes in the result. Furthermore, `xsl:pi` may not include any instructions or literal text that inserts a `?>` in the output since that would prematurely end the processing instruction.

Generating Comments with `xsl:comment`

The `xsl:comment` element inserts a comment in the output document. It has no attributes. Its contents are the text of the comment. For example,

```
<xsl:template select="ATOM">
  <xsl:comment>There was an atom here once.</xsl:comment>
</xsl:template>
```

This rule replaces `ATOM` nodes with this output:

```
<!--There was an atom here once.-->
```

The contents of the `xsl:comment` element can include `xsl:value-of` elements and `xsl:apply-templates` elements provided the result of these instructions is pure text. It may not contain `xsl:element` and other instructions that produce elements and attributes in the result. Furthermore, `xsl:comment` may not include any instructions or literal text that inserts a double hyphen in the comment. This would result in a malformed comment in the output, which is forbidden.

Generating Text with `xsl:text`

The `xsl:text` element inserts its contents into the output document as literal text. For example, this rule replaces each `ATOM` element with the string “There was an atom here once.”

```
<xsl:template select="ATOM">
  <xsl:text>There was an atom here once.</xsl:text>
</xsl:template>
```

The `xsl:text` element isn't much used because most of the time it's easier to simply type the text. However, `xsl:text` does have one advantage. It preserves whitespace exactly. This is useful when dealing with poetry, computer source code, or other information where whitespace is significant.

Copying the Current Node with `xsl:copy`

The `xsl:copy` element copies the source node into the output. Child elements, attributes, and other content are not automatically copied. However, the contents of the `xsl:copy` element are an `xsl:template` element that can select these things to be copied as well. This is often useful when transforming a document from one markup vocabulary to the same or a closely related markup vocabulary. For example, this template rule strips the attributes and child elements off an atom and replaces it with the value of its contents:

```
<xsl:template match="ATOM">
  <xsl:copy>
    <xsl:apply-templates/>
  </xsl:copy>
</xsl:template>
```

One useful template `xsl:copy` makes possible is the identity transformation; that is, a transformation from a document into itself. Such a transformation looks like this:

```
<xsl:template match="*|@*|comment()|pi()|text()">
  <xsl:copy>
    <xsl:apply-templates select="*|@*|comment()|pi()|text()"/>
  </xsl:copy>
</xsl:template>
```

You can adjust the identity transformation a little to produce similar documents. For example, Listing 14-15 is a style sheet that strips comments from a document, leaving the document otherwise untouched. It resulted from leaving the `comment()` node out of the `match` and `select` attribute values in the identity transformation.

Listing 14-15: An XSL style sheet that strips comments from a document

```
<?xml version="1.0"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/XSL/Transform/1.0">

  <xsl:template match="*|@*|pi()|text()">
    <xsl:copy>
```

```

        <xsl:apply-templates select="*|@*|pi()|text()"/>
      </xsl:copy>
    </xsl:template>

  </xsl:stylesheet>

```

`xsl:copy` only copies the source node. You can copy other nodes, possibly more than one of them, using `xsl:copy-of`. The `select` attribute of `xsl:copy-of` chooses the nodes to be copied. For example, Listing 14-16 is a stylesheet that uses `xsl:copy-of` to strip out elements without melting points from the periodic table by copying only `ATOM` elements that have `MELTING_POINT` children.

Listing 14-16: A stylesheet that copies only `ATOM` elements that have `MELTING_POINT` children

```

<?xml version="1.0"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/XSL/Transform/1.0">

  <xsl:template match="/PERIODIC_TABLE">
    <PERIODIC_TABLE>
      <xsl:apply-templates select="ATOM"/>
    </PERIODIC_TABLE>
  </xsl:template>

  <xsl:template match="ATOM">
    <xsl:apply-templates
      select="MELTING_POINT"/>
  </xsl:template>

  <xsl:template match="MELTING_POINT">
    <xsl:copy-of select="..">
      <xsl:apply-templates select="*|@*|pi()|text()"/>
    </xsl:copy-of>
  </xsl:template>

  <xsl:template match="*|@*|pi()|text()">
    <xsl:copy>
      <xsl:apply-templates select="*|@*|pi()|text()"/>
    </xsl:copy>
  </xsl:template>

</xsl:stylesheet>

```

**Note**

This is an example of an XSL transformation from a source vocabulary to the same vocabulary. Unlike most of the examples in this chapter, it does not transform to well-formed HTML.

Counting Nodes with `xsl:number`

The `xsl:number` element inserts a formatted integer into the output document. The value of the integer is given by rounding the number calculated by the `expr` attribute to the nearest integer, then formatting it according to the value of the `format` attribute. Reasonable defaults are provided for both these attributes. For example, consider the style sheet for the `ATOM` elements in Listing 14-17.

Listing 14-17: An XSL style sheet that counts atoms

```
<?xml version="1.0"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/XSL/Transform/1.0">

  <xsl:template match="PERIODIC_TABLE">
    <html>
      <head><title>The Elements</title></head>
      <body>
        <table>
          <xsl:apply-templates select="ATOM"/>
        </table>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="ATOM">
    <tr>
      <td><xsl:number expr="position()"/></td>
      <td><xsl:value-of select="NAME"/></td>
    </tr>
  </xsl:template>

</xsl:stylesheet>
```

When this style sheet is applied to Listing 14-1, the output appears like this:

```
<html><head><title>The
Elements</title></head><body><table><tr><td>1</td><td>Hydrogen<
/td></tr>
```

```
<tr><td>2</td><td>Helium</td></tr>
</table></body></html>
```

Hydrogen gets number 1 because it is the first `ATOM` element in its parent. Helium gets number 2 because it is the second `ATOM` element in its parent. (That these are the atomic numbers of hydrogen and helium is a side effect of Listing 14-1 being arranged in order of atomic number.)

Default Numbers

If you use the `expr` attribute to calculate the number, that's all you need. However, if the `expr` attribute is omitted, then the position of the current node in the source tree is used as the number. However, this default can be adjusted using these three attributes:

- ♦ `level`
- ♦ `count`
- ♦ `from`



These three attributes are a holdover from previous drafts of XSL that did not support the more complex expressions now possible. If they seem at all confusing to you, I recommend that you ignore them and use `expr` instead.

The level Attribute

By default, with no `expr` attribute, `xsl:number` counts sibling nodes of the source node. For instance, if the `ATOMIC_NUMBER` elements were numbered instead of `ATOM` elements, none would have a number higher than 1 because an `ATOM` never has more than one `ATOMIC_NUMBER` child. Although the document contains more than one `ATOMIC_NUMBER` element, these are not siblings.

Setting the `level` attribute of `xsl:number` to any counts all of the elements of the same kind as the current node in the document. This includes not just the ones that may match the current rule, but all elements of the right type. Even if you select only the atomic numbers of the gases, for example, the solids and liquids would still count, even if they weren't output. Consider, these rules:

```
<xsl:template match="ATOM">
  <xsl:apply-templates select="NAME"/>
</xsl:template>

<xsl:template match="NAME">
  <td><xsl:number level="any"/></td>
  <td><xsl:value-of select="."/></td>
</xsl:template>
```

Since `level` is set to `any`, they produce output that doesn't start from 1 with each new `NAME` element like this:

```
<td>1</td><td>Hydrogen</td>

<td>2</td><td>Helium</td>
```

If you remove the `level` attribute or set it to its default value of `single`, then the output would look like this:

```
<td>1</td><td>Hydrogen</td>

<td>1</td><td>Helium</td>
```

A slightly less useful option sets the `level` attribute of `xsl:number` to `multi` to specify that both the siblings of the current node and its ancestors (but not their children that aren't siblings of the current node) should be counted.

The count Attribute

By default, with no `expr` attribute, only elements of the same type as the element of the current node get counted. However, you can set the `count` attribute of `xsl:number` to a `select` expression that specifies what to count. For instance, this rule applies numbers all the child elements of an `ATOM`:

```
<xsl:template match="ATOM/*">
  <td><xsl:number count="*" /></td>
  <td><xsl:value-of select="." /></td>
</xsl:template>
```

The output from applying this rule looks like this:

```
<td>1</td><td>Helium</td>
<td>2</td><td>He</td>
<td>3</td><td>2</td>
<td>4</td><td>4.0026</td>
<td>5</td><td>1</td>
<td>6</td><td>4.216</td>
<td>7</td><td>0.95</td>
<td>8</td><td>
  0.1785
</td>
```

The from Attribute

The `from` attribute contains a `select` expression that specifies which element the counting begins with in the input tree. However, the counting still begins from one, not two or ten or some other number. The `from` attribute only changes which element is considered to be the first element.

Number to String Conversion

Until now, I've implicitly assumed that numbers looked like 1, 2, 3, and so on; that is, a European numeral starting from 1 and counting by 1. However, that's not the only possibility. For instance, the pages of the preface and other front matter of books often appear in small Roman numerals like i, ii, iii, iv, and so on. And different countries use different conventions to group the digits, separate the integer and fractional parts of a real number, and represent the symbols for the various digits. These are all adjustable through five attributes of `xsl:number`:

- ♦ `format`
- ♦ `letter-value`
- ♦ `digit-group-sep`
- ♦ `n-digits-per-group`
- ♦ `sequence-src`

The `format` Attribute

You can adjust the numbering style used by `xsl:number` using the `format` attribute. This attribute generally has one of the following values:

- ♦ `i`: produces the sequence of lowercase Roman numerals i, ii, iii, iv, v, vi, ...
- ♦ `I`: produces the sequence of uppercase Roman numerals I, II, III, IV, V, VI, ...
- ♦ `a`: produces the sequence of lowercase letters a, b, c, d, e, f, ...
- ♦ `A`: produces the sequence of uppercase letters A, B, C, D, E, F, ...

For example, this rule numbers the atoms with capital Roman numerals:

```
<xsl:template match="ATOM">
  <P>
    <xsl:number expr="position()" format="I"/>
    <xsl:value-of select="."/>
  </P>
</xsl:template>
```

You can adjust the number (or letter) at which counting starts by changing the value of the `format` attribute. For example, to start numbering at 5, set `format="5"`. To start numbering at iii, set `format="iii"`.

You can also specify decimal numbering with leading zeroes by including the number of leading zeroes you want in the `format` attribute. For instance, setting `format="01"`, produces the sequence 01, 02, 03, 04, 05, 06, 07, 08, 09, 10, 11, 12, You might find this useful when lining numbers up in columns.

The letter-value Attribute

The `letter-value` attribute distinguishes between letters interpreted as numbers and letters interpreted as letters. For instance, if you want to use `format="I"` to start the sequence I, J, K, L, M, N, ... instead of I, II, III, IV, V, VI, ... you would set the `letter-value` attribute to the keyword `alphabetic`. The keyword `other` specifies a numeric sequence. For example,

```
<xsl:template match="ATOM">
  <P>
    <xsl:number expr="position()"
                format="I" letter-value="alphabetic"/>
    <xsl:value-of select="."/>
  </P>
</xsl:template>
```

Group Separator Attributes

In the United States, we tend to write large numbers with commas grouping every three digits like 4,567,302,000. However, in many languages and countries, a period or a space separates the groups instead; for instance, 4.567.302.000 or 4 567 302 000. Furthermore, in some countries it's customary to group large numbers every four digits instead of every three; for example, 4,5673,0000. If you're dealing with very long lists that may contain a thousand or more items, you need to worry about these issues.

The `digit-group-sep` attribute specifies the grouping separator used between groups of digits. The `n-digits-per-group` attribute specifies the number of digits used in a group. Generally, you'd make these attributes contingent on the language. For example,

```
<xsl:number digit-group-sep=" "/>
```

The sequence-src Attribute

Finally, if you want to use an unusual order (for example, a list of date strings like 1-1-1999, 1-2-1999, 1-3-1999, ... or a list that jumps by tens like 10, 20, 30, 40, ...) you can store this list (separated by whitespace) in a separate document. The `sequence-src` attribute has a value representing the relative or absolute URL of this document. For example,

```
<xsl:number sequence-src="1999.txt"/>
```

Sorting Output Elements

The `xsl:sort` element sorts the output elements into a different order than they appear in the input. An `xsl:sort` element appears as a child of an `xsl:apply-`

templates **element** or **xsl:for-each element**. The **select** attribute of the **xsl:sort element** defines the key used to sort the elements' output by **xsl:apply-templates** or **xsl:for-each**.

By default, sorting is performed in alphabetical order of the keys. If more than one **xsl:sort element** is present in a given **xsl:apply-templates** or **xsl:for-each element**, then the output sorts first by the first key, then by the second key, and so on. If any elements still compare equally, they output in the order they appear in the source document.

For example, suppose you have a file full of **ATOM** elements arranged alphabetically. To sort by atomic number, you can use the style sheet in Listing 14-18.

Listing 14-18: An XSL style sheet that sorts by atomic number

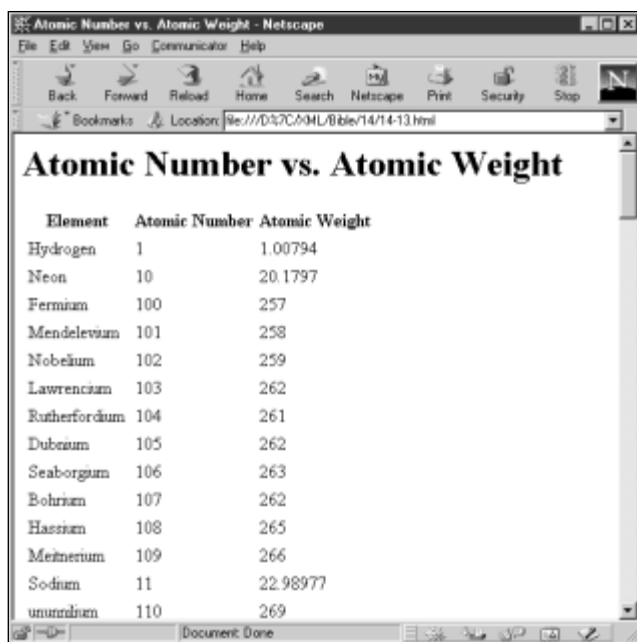
```
<?xml version="1.0"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/XSL/Transform/1.0">

  <xsl:template match="PERIODIC_TABLE">
    <html>
      <head>
        <title>Atomic Number vs. Atomic Weight</title>
      </head>
      <body>
        <h1>Atomic Number vs. Atomic Weight</h1>
        <table>
          <th>Element</th>
          <th>Atomic Number</th>
          <th>Atomic Weight</th>
          <xsl:apply-templates>
            <xsl:sort select="ATOMIC_NUMBER"/>
          </xsl:apply-templates>
        </table>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="ATOM">
    <tr>
      <td><xsl:apply-templates select="NAME"/></td>
      <td><xsl:apply-templates select="ATOMIC_NUMBER"/></td>
      <td><xsl:apply-templates select="ATOMIC_WEIGHT"/></td>
    </tr>
  </xsl:template>

</xsl:stylesheet>
```

Figure 14-5 shows the results that display the limits of alphabetical sorting. Hydrogen, atomic number 1, is the first element. However, the second element is not helium, atomic number 2, but rather neon, atomic number 10. Although 10 sorts after 9 numerically, alphabetically 10 falls before 2.



Element	Atomic Number	Atomic Weight
Hydrogen	1	1.00794
Neon	10	20.1797
Fermium	100	257
Mendelevium	101	258
Nobelium	102	259
Lawrencium	103	262
Rutherfordium	104	261
Dubnium	105	262
Seaborgium	106	263
Bohrium	107	262
Hassium	108	265
Meitnerium	109	266
Sodium	11	22.98977
ununbium	110	269

Figure 14-5: Atoms alphabetically sorted by atomic number

You can, however, adjust the order of the sort by setting the optional `data-type` attribute to the value `number`. For example,

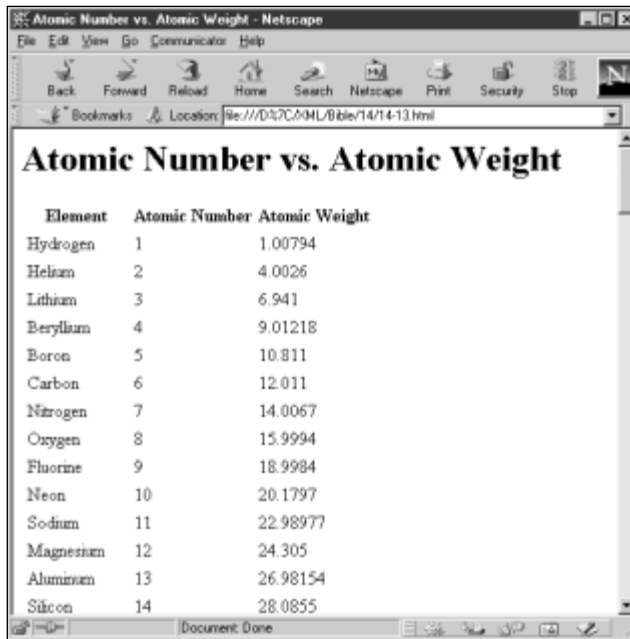
```
<xsl:sort data-type="number" select="ATOMIC_NUMBER"/>
```

Figure 14-6 shows the elements sorted properly.

You can change the order of the sort from the default ascending order to descending by setting the `order` attribute to `descending` like this:

```
<xsl:sort order="descending"
  sort="number"
  select="ATOMIC_NUMBER"/>
```

This sorts the elements from the largest atomic number to the smallest so that hydrogen now appears last in the list.



Element	Atomic Number	Atomic Weight
Hydrogen	1	1.00794
Helium	2	4.0026
Lithium	3	6.941
Beryllium	4	9.01218
Boron	5	10.811
Carbon	6	12.011
Nitrogen	7	14.0067
Oxygen	8	15.9994
Fluorine	9	18.9984
Neon	10	20.1797
Sodium	11	22.98977
Magnesium	12	24.305
Aluminum	13	26.98154
Silicon	14	28.0855

Figure 14-6: Atoms numerically sorted by atomic number

Alphabetical sorting naturally depends on the alphabet. The `lang` attribute can set the language of the keys. The value of this attribute should be an ISO 639 language code like `en` for English.

Cross-Reference

These are the same values supported by the `xml:lang` attribute discussed in Chapter 10, *Attribute Declarations in DTDs*.

Finally, you can set the `case-order` attribute to one of the two values `upper-first` or `lower-first` to specify whether uppercase letters sort before lowercase letters or vice versa. The default depends on the language.

CDATA and < Signs

Standard XSL contains no means to insert raw, unescaped `<` characters that are not part of a tag into the output. Raw less-than signs make the output document malformed, something XSL does not allow. Instead, if you use a character reference like `<`; or the entity reference `<`; to insert the `<` character, the formatter will insert `<`; or perhaps `<`;

This only really becomes important when you're embedding JavaScript into a page because JavaScript uses the `<` character to mean numerical less than rather than the start of the tag. Furthermore, JavaScript implementations do not allow the `<` character to be replaced with the HTML entity `<`;

You can insert raw, unescaped `>` and `>=` signs into the output, however. Consequently, if your output needs to contain JavaScript that makes numerical comparisons, you can rewrite a less-than comparison as a greater-than-or-equal-to comparison by reversing the order of the operands. You can rewrite a less-than-or-equal-to comparison as a greater-than comparison. For example, here's a few lines of JavaScript code I use in a lot of my Web pages:

```
if (location.host.toLowerCase().indexOf("sunsite") < 0) {  
    location.href="http://metalab.unc.edu/xml/";  
}
```

These lines are malformed because of the less-than sign in the first two lines. However, these lines are completely equivalent to these lines:

```
if (0 > location.host.toLowerCase().indexOf("sunsite")) {  
    location.href="http://metalab.unc.edu/xml/";  
}
```

If you have multiple tests combined with Boolean operators, you may need to change logical ands to logical ors as well. For example, these two lines of JavaScript effectively test whether the location of the page is not on metalab and not on sunsite:

```
if (location.host.toLowerCase().indexOf("metalab") < 0  
    && location.host.toLowerCase().indexOf("sunsite") < 0) {  
    location.href="http://metalab.unc.edu/xml/";  
}
```

These lines are malformed because of the less-than signs in the first two lines. However, these lines which test whether the page is on metalab or sunsite are completely equivalent:

```
if (0 > location.host.toLowerCase().indexOf("metalab")  
    || 0 > location.host.toLowerCase().indexOf("sunsite")) {  
    location.href="http://metalab.unc.edu/xml/";  
}
```

Tip

You can also place the offending JavaScript in a separate document, and link to it from the `SCRIPT` element's `SRC` attribute. However, this is unreliable prior to Internet Explorer 4 and Netscape Navigator 3.

CDATA sections are not allowed in output for reasons of simplicity. A CDATA section can always be replaced by an equivalent collection of character data with

Unicode escapes for problem characters like < and &. CDATA sections are purely a convenience for humans writing XML files by hand. Computer programs like XSL formatters don't need them.



Note

The XSL formatter included in Internet Explorer 5.0 does support a non-standard `xsl:cdata` element for inserting CDATA sections into the output. However, it's unlikely that this will be added to standard XSL, and it may even be removed from later releases of Internet Explorer.

Modes

Sometimes you want to include the same content from the source document in the output document multiple times. That's easy to do by simply applying templates multiple times, once in each place where you want the data to appear. However, suppose you want the data to be formatted differently in different locations? That's a little trickier.

For example, suppose you want the output of processing the periodic table to be a series of 100 links to more detailed descriptions of the individual atoms. In this case, the output document would start like this:

```
<UL>
<LI><A HREF="#Ac">Actinium</A></LI>
<LI><A HREF="#Al">Aluminum</A></LI>
<LI><A HREF="#Am">Americium</A></LI>
<LI><A HREF="#Sb">Antimony</A></LI>
<LI><A HREF="#Ar">Argon</A></LI>
...
```

Later in the document, the actual atom description would appear, formatted like this:

```
<H3><A NAME="Al">Aluminum</A></H3><P>
  Aluminum
  26.98154
  13
  3
  2740
  933.5
  Al

  2.7

</P>
```

This sort of application is common anytime you automatically generate a hypertext table of contents or index. The `NAME` of the atom must be formatted

differently in the table of contents than in the body of the document. You need two different rules that both apply to the `ATOM` element at different places in the document. The solution is to give each of the different rules a `mode` attribute. Then you can choose which template to apply by setting the `mode` attribute of the `xsl:apply-templates` element. Listing 14-19 demonstrates.

Listing 14-19: An XSL stylesheet that uses modes to format the same data differently in two different places

```
<?xml version="1.0"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/XSL/Transform/1.0">

  <xsl:template match="/PERIODIC_TABLE">
    <HTML>
      <HEAD><TITLE>The Elements</TITLE></HEAD>
      <BODY>

        <H2>Table of Contents</H2>
        <UL>
          <xsl:apply-templates select="ATOM" mode="toc"/>
        </UL>

        <H2>The Elements</H2>
        <xsl:apply-templates select="ATOM" mode="full"/>

      </BODY>
    </HTML>
  </xsl:template>

  <xsl:template match="ATOM" mode="toc">
    <LI><A>
      <xsl:attribute name="HREF">#<xsl:value-of
        select="SYMBOL"/></xsl:attribute>
      <xsl:value-of select="NAME"/>
    </A></LI>
  </xsl:template>

  <xsl:template match="ATOM" mode="full">
    <H3><A>
      <xsl:attribute name="NAME">
        <xsl:value-of select="SYMBOL"/>
      </xsl:attribute>
      <xsl:value-of select="NAME"/>
    </A></H3>
    <P>
      <xsl:value-of select="."/>
    </P>
  </xsl:template>
</xsl:stylesheet>
```

```
</xsl:template>

</xsl:stylesheet>
```

Defining Constants with `xsl:variable`

Named constants help clean up code. They can replace commonly used boilerplate text with a simple name and reference. They can also make it easy to adjust boilerplate text that appears in multiple locations by simply changing the constant definition.

The `xsl:variable` element defines a named string for use elsewhere in the style sheet via an attribute value template. The `xsl:variable` element is an empty element that appears as a direct child of `xsl:stylesheet`. It has a single attribute, `name`, which provides a name by which the variable can be referred to. The contents of the `xsl:variable` element provides the replacement text. For example, this `xsl:variable` element defines a variable with the name `copy99` and the value Copyright 1999 Elliotte Rusty Harold:

```
<xsl:variable name="copy99">
  Copyright 1999 Elliotte Rusty Harold
</xsl:variable>
```

To access the value of this variable, you prefix a dollar sign to the name of the variable. To insert this in an attribute, use an attribute value template. For example:

```
<BLOCK COPYRIGHT="{ $copy99 }">
</BLOCK >
```

You can use `xsl:value-of` to insert the variable's replacement text into the output document as text:

```
<xsl:value-of select="$copy99"/>
```

The contents of the `xsl:variable` can contain markup including other XSL instructions. This means that you can calculate the value of a variable based on other information, including the value of other variables. However, a variable may not refer to itself recursively, either directly or indirectly. For instance, the following example is in error:

```
<xsl:variable name="GNU">
  <xsl:value-of select="$GNU"/>'s not Unix
</xsl:variable>
```

Similarly, two variables may not refer to each other in a circular fashion like this:

```
<xsl:variable name="Thing1">
  Thing1 loves <xsl:value-of select="$Thing2"/>
</xsl:variable>

<xsl:variable name="Thing2">
  Thing2 loves <xsl:value-of select="$Thing1"/>
</xsl:variable>
```

Named Templates

Variables are limited to basic text and markup. XSL provides a more powerful macro facility that can wrap standard markup and text around changing data. For example, suppose you want an atom's atomic number, atomic weight, and other key values format as a table cell in small, bold type Times in blue. In other words, you want the output to look like this:

```
<td>
  <font face="Times, serif" color="blue" size="2">
    <b>52</b>
  </font>
</td>
```

You can certainly include all that in a template rule like this:

```
<xsl:template match="ATOMIC_NUMBER">
  <td>
    <font face="Times, serif" color="blue" size="2">
      <b>
        <xsl:value-of select="."/>
      </b>
    </font>
  </td>
</xsl:template>
```

This markup can repeat as the template of other rules, or as a part of the template used in other rules. When the detailed markup grows more complex and when it appears in several different places in a style sheet, you may elect to turn it into a named template. Named templates resemble variables. However, they enable you to include data from the place where the template is applied, rather than merely inserting fixed text.

The `xsl:template` element can have a `name` attribute by which it can be explicitly invoked, even when it isn't applied indirectly. For example, this shows a sample named template for the above pattern:

```
<xsl:template name="ATOM_CELL">
  <td>
```

```

        <font face="Times, serif" color="blue" size="2">
            <b>
                <xsl:value-of select="."/>
            </b>
        </font>
    </td>
</xsl:template>

```

The `<xsl:value-of select="."/>` element in the middle of the macro will be replaced by the contents of the current node from which this template was called.

The `xsl:call-template` element appears in the contents of a template rule. It has a required `name` argument that names the template it will call. When processed, the `xsl:call-template` element is replaced by the contents of the `xsl:template` element it names. For example, we can now rewrite the `ATOMIC_NUMBER` rule like this using the `xsl:call-template` element to call the `ATOM_CELL` named template:

```

<xsl:template match="ATOMIC_NUMBER">
    <xsl:call-template name="ATOM_CELL"/>
</xsl:template>

```

This fairly simple example only saves a few lines of code, but the more complicated the template, and the more times it's reused, the greater the reduction in complexity of the style sheet. Named templates also have the advantage, like variables, of factoring out common patterns in the style sheet so you can edit them as one. For instance, if you decide to change the color of atomic number, atomic weight, and other key values from blue to red, you only need to change it once in the named template. You do not have to change it in each separate template rule. This facilitates greater consistency of style in the long run.

Parameters

Each separate invocation of a named template can pass parameters to the template to customize its output. In the `xsl:template` element, the parameters are represented as `xsl:param-variable` child elements. In the `xsl:call-template` element, parameters are represented as `xsl:param` child elements.

For example, suppose you want to also include a link to a particular file for each atom cell. The output should look something like this:

```

<td>
    <font face="Times, serif" color="blue" size="2">
        <b>
            <a href="atomic_number.html">52</a>
        </b>
    </font>
</td>

```

The trick is that the value of the `href` attribute has to be passed in from the point where the template is invoked because it changes for each separate invocation of the template. For example, atomic weights will have to format like this:

```
<td>
  <font face="Times, serif" color="blue" size="2">
    <b>
      <a href="atomic_weight.html">4.0026</a>
    </b>
  </font>
</td>
```

The template that supports this looks like this:

```
<xsl:template name="ATOM_CELL">
  <xsl:param-variable name="file">
    index.html
  </xsl:param-variable>
  <td>
    <font face="Times, serif" color="blue" size="2">
      <b>
        <a href="{ $file }"><xsl:value-of select="."/></a>
      </b>
    </font>
  </td>
</xsl:template>
```

The `name` attribute of the `xsl:param-variable` element gives the parameter a name (important if there are multiple arguments) and the contents of the `xsl:param-variable` element supplies a default value for this parameter to be used if the invocation doesn't provide a value. (This can also be given as a string expression using an `expr` attribute, exactly like `xsl:variable`.)

When this template is called, an `xsl:param` child of the `xsl:call-template` element provides the value of the parameter using its `name` attribute to identify the parameter and its contents to provide a value for the parameter. For example:

```
<xsl:template match="ATOMIC_NUMBER">
  <xsl:call-template macro="ATOM_CELL">
    <xsl:param name="file">atomic_number.html</xsl:param>
    <xsl:value-of select="."/>
  </xsl:call-template>
</xsl:template>
```

Again, this is a fairly simple example. However, much more complex named templates exist. For instance, you could define header and footer macros for pages on a Web site for importing by many different style sheets, each of which would only have to change a few parameters for the name of the page author, the title of the page, and the copyright date.

Stripping and Preserving Whitespace

You may have noticed that all the examples of output up till now have been formatted a little strangely. The reason the examples appeared strange is that the source document needed to break long lines across multiple lines to fit between the margins of this book. Unfortunately, the extra whitespace added to the input document carried over into the output document. For a computer, the details of insignificant whitespace aren't important, but for a person they can be distracting.

The default behavior for text nodes like the content of an `ATOMIC_NUMBER` or `DENSITY` element is to preserve all whitespace. A typical `DENSITY` element looks like this:

```
<DENSITY UNITS="grams/cubic centimeter"><!-- At 300K -->
  7.9
</DENSITY>
```

When its value is taken the leading and trailing whitespace is included, like this, even though it's really only there to help fit on this printed page and isn't at all significant:

```
7.9
```

However, there is one exception. If the text node contains only whitespace, no other text, then the space is considered insignificant and stripped. But there is one exception to the exception: if the text has an ancestor with an `xml:space` attribute with the value `preserve`, then it is not stripped unless a closer ancestor contains an `xml:space` attribute with the value `default`. (This is really simpler than it sounds. All this says is that you can ignore text nodes that contain only whitespace unless they're specifically set to have significant whitespace. Otherwise whitespace is preserved.)

If none of the elements in the document should preserve whitespace, then you can set the `default-space` attribute of the `xsl:stylesheet` element to `strip` and all leading and trailing whitespace will be removed from text nodes before they're output. This is the easiest solution for the periodic table problem. For example:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/XSL/Transform/1.0"
  default-space="strip">
```

If you don't want to strip space from all elements, you can use `xsl:strip-space` elements to identify the specific elements in the input document whose whitespace should be considered insignificant and not copied to the output document. The `element` attribute identifies the element whose excess space should be trimmed.

For example, these rules could be added to the periodic table style sheet to avoid excess whitespace:

```
<xsl:strip-space element="DENSITY"/>
<xsl:strip-space element="BOILING_POINT"/>
<xsl:strip-space element="MELTING_POINT"/>
```

The `xsl:preserve-space` element is the opposite of the `xsl:strip-space` element. Its `element` attribute names an element whose whitespace should be preserved. For example:

```
<xsl:preserve-space element="ATOM"/>
```

Whitespace in the style sheet itself (as opposed to whitespace in the input XML document) is not considered significant and is reduced to a single space by default. You can avoid this behavior only by enclosing the literal whitespace in an `xsl:text` element. For example:

```
<xsl:template select="ATOM">
  <xsl:text>    This is indented exactly five spaces.</xsl:text>
</xsl:template>
```

One final trick you can play with whitespace is to attach an `indent-result` attribute to the root `xsl:stylesheet` element. If this attribute has the value `yes`, then the processor is allowed to inset (but not remove) extra whitespace into the output to try to “pretty-print” the output. This may include indentation and line breaks. For example:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/XSL/Transform/1.0"
  indent-result="yes">
  <!-- usual templates and such go here... -->
</xsl:stylesheet>
```

If you’re generating HTML, specifying `indent-result="yes"` can only make the output more readable. The default value of `indent-result` is `no` because other, non-HTML output formats may consider whitespace more significant.

Making Choices

XSL provides two elements that allow you to change the output based on the input. The `xsl:if` element either does or does not output a given fragment of XML depending on what patterns are present in the input. The `xsl:choose` element picks one of several possible XML fragments depending on what patterns are present in the input. Most of what you can do with `xsl:if` and `xsl:choose` can also be done by suitable application of templates. However, sometimes the solution with `xsl:if` or `xsl:choose` is simpler and more obvious.

xsl:if

The `xsl:if` element provides a simple facility for changing the output based on a pattern. The `test` attribute of `xsl:if` contains a `select` expression that evaluates to a Boolean. If the expression is true, the contents of the `xsl:if` element are output. Otherwise, they're not. For example, this template writes out the names of all `ATOM` elements. A comma and a space is added after all except the last element in the list.

```
<xsl:template match="ATOM">
  <xsl:value-of select="NAME"/>
  <xsl:if test="not(position()=last())">, </xsl:if>
</xsl:template>
```

This ensures that the list looks like “Hydrogen, Helium” and not “Hydrogen, Helium”.

There are no `xsl:else` or `xsl:else-if` elements. The `xsl:choose` element provides this functionality.

xsl:choose

The `xsl:choose` element selects one of several possible outputs depending on several possible conditions. Each condition and its associated output template is provided by an `xsl:when` child element. The `test` attribute of the `xsl:when` element is a `select` expression with a Boolean value. If multiple conditions are true, only the first true one is instantiated. If none of the `xsl:when` elements are true, the contents of the `xsl:otherwise` child elements are instantiated. For example, this rule changes the color of the output based on whether the `STATE` attribute of the `ATOM` element is `SOLID`, `LIQUID`, or `GAS`:

```
<xsl:template match="ATOM">
  <xsl:choose>
    <xsl:when test="@STATE='SOLID'">
      <P style="color:black">
        <xsl:value-of select="."/>
      </P>
    </xsl:when>
    <xsl:when test="@STATE='LIQUID'">
      <P style="color:blue">
        <xsl:value-of select="."/>
      </P>
    </xsl:when>
    <xsl:when test="@STATE='GAS'">
      <P style="color:red">
        <xsl:value-of select="."/>
      </P>
    </xsl:when>
    <xsl:otherwise>
      <P style="color:green">
        <xsl:value-of select="."/>
      </P>
    </xsl:otherwise>
  </xsl:choose>
```

```
    </xsl:other>
  </xsl:choose>
</xsl:template>
```

Merging Multiple Style Sheets

A single XML document may use many different markup vocabularies described in many different DTDs. You may also wish to use different standard style sheets for those different vocabularies. However, you'll also want style rules for particular documents as well. The `xsl:import` and `xsl:include` elements enable you to merge multiple style sheets so that you can organize and reuse style sheets for different vocabularies and purposes.

Import with `xsl:import`

The `xsl:import` element is a top level element whose `href` attribute provides the URI of a style sheet to import. All `xsl:import` elements must appear before any other top level elements in the `xsl:stylesheet` root element. For example, these `xsl:import` elements import the style sheets `genealogy.xml` and `standards.xml`.

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/XSL/Transform/1.0">
  <xsl:import href="genealogy.xml"/>
  <xsl:import href="standards.xml"/>
  <!-- other child elements follow -->
</xsl:stylesheet>
```

Rules in the imported style sheets may conflict with rules in the importing style sheet. If so, rules in the importing style sheet take precedence. If two rules in different imported style sheets conflict, then the last one imported (`standards.xml` above) takes precedence.

The `xsl:apply-imports` element is a slight variant of `xsl:apply-templates` that only uses imported rules. It does not use any rules from the importing style sheet. This allows access to imported rules that would otherwise be overridden by rules in the importing style sheet. Other than the name, it has identical syntax to `xsl:apply-templates`. The only behavioral difference is that it only matches template rules in imported style sheets.

Inclusion with `xsl:include`

The `xsl:include` element is a top level element that copies another style sheet into the current style sheet at the point where it occurs. (More precisely, it copies the contents of the `xsl:stylesheet` element in the remote document into the current document.) Its `href` attribute provides the URI of the style sheet to include. An `xsl:include` element can occur anywhere at the top level after the last `xsl:import` element.

Unlike rules included by `xsl:import` elements, rules included by `xsl:include` elements have the same precedence in the including style sheet that they would have if they were copied and pasted from one stylesheet to the other. As far as the formatting engine is concerned, there is no difference between an included rule and a rule that's physically present.

Embed Style Sheets in Documents with `xsl:stylesheet`

You can directly include an XSL style sheet in the XML document it applies to. I don't recommend this in practice, and browsers and formatting engines are not required to support it. Nonetheless, a few might. To do this, the `xsl:stylesheet` element must appear as a child of the document element, rather than as a root element itself. It would have an `id` attribute giving it a unique name, and this `id` attribute would appear as the value of the `href` attribute in the `xml-stylesheet` processing instruction, following the anchor identifier `#`. Listing 14-20 demonstrates:

Listing 14-20: An XSL style sheet embedded in an XML document

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="#id(mystyle)"?>
<PERIODIC_TABLE>

  <xsl:stylesheet
    xmlns:xsl="http://www.w3.org/XSL/Transform/1.0"
    id="mystyle">

    <xsl:template match="/">
      <html>
        <xsl:apply-templates/>
      </html>
    </xsl:template>

    <xsl:template match="PERIODIC_TABLE">
      <xsl:apply-templates/>
    </xsl:template>

    <xsl:template match="ATOM">
      <P>
        <xsl:value-of select="."/>
      </P>
    </xsl:template>

  </xsl:stylesheet>
```

Continued

Listing 14-20 (continued)

```

<ATOM>
  <NAME>Actinium</NAME>
  <ATOMIC_WEIGHT>227</ATOMIC_WEIGHT>
  <ATOMIC_NUMBER>89</ATOMIC_NUMBER>
  <OXIDATION_STATES>3</OXIDATION_STATES>
  <BOILING_POINT UNITS="Kelvin">3470</BOILING_POINT>
  <MELTING_POINT UNITS="Kelvin">1324</MELTING_POINT>
  <SYMBOL>Ac</SYMBOL>
  <DENSITY UNITS="grams/cubic centimeter"><!-- At 300K -->
    10.07
  </DENSITY>
  <ELECTRONEGATIVITY>1.1</ELECTRONEGATIVITY>
  <ATOMIC_RADIUS UNITS="Angstroms">1.88</ATOMIC_RADIUS>
</ATOM>

</PERIODIC_TABLE>

```

Summary

In this chapter, you learned about XSL transformations. In particular, you learned the following:

- ♦ The Extensible Style Language (XSL) comprises two separate XML applications for transforming and formatting XML documents.
- ♦ An XSL transformation applies rules to a tree read from an XML document to transform it into an output tree written as an XML document.
- ♦ An XSL template rule is an `xsl:template` element with a `match` attribute. Nodes in the input tree are compared against the patterns of the `match` attributes of the different template elements. When a match is found, the contents of the template are output.
- ♦ The value of a node is a pure text (no markup) string containing the contents of the node. This can be calculated by the `xsl:value-of` element.
- ♦ You can process multiple elements in two ways: the `xsl:apply-templates` element and the `xsl:for-each` element.
- ♦ The value of the `match` attribute of the `xsl:template` element is a match pattern specifying which nodes the template matches.
- ♦ Select expressions are a superset of the `match` attribute used by the `select` attribute of `xsl:apply-templates`, `xsl:value-of`, `xsl:for-each`, `xsl:copy-of`, and `xsl:sort` and various other elements.

- ♦ Two default rules apply templates to element nodes and take the value of text nodes.
- ♦ The `xsl:element`, `xsl:attribute`, `xsl:pi`, `xsl:comment`, and `xsl:text` elements can output elements, attributes, processing instructions, comments, and text calculated from data in the input document.
- ♦ The `xsl:attribute-set` element defines a common group of attributes that can be applied to multiple elements in different templates with the `xsl:use` element.
- ♦ The `xsl:copy` element copies the current input node into the output.
- ♦ The `xsl:number` element inserts the number specified by its `expr` attribute into the output using a specified number format given by the `format` attribute.
- ♦ The `xsl:sort` element can reorder the input nodes before copying them to the output.
- ♦ XSL cannot output CDATA sections or unescaped `<` signs.
- ♦ Modes can apply different templates to the same element from different locations in the style sheet.
- ♦ The `xsl:variable` element defines named constants that can clarify your code.
- ♦ Named templates help you reuse common template code.
- ♦ Whitespace is maintained by default unless an `xsl:strip-space` element or `xml:space` attribute says otherwise.
- ♦ The `xsl:if` element produces output if, and only if, its `test` attribute is true.
- ♦ The `xsl:choose` element outputs the template of the first one of its `xsl:when` children whose `test` attribute is true, or the template of its `xsl:default` element if no `xsl:when` element has a true `test` attribute.
- ♦ The `xsl:import` and `xsl:include` elements merge rules from different style sheets.

In the next chapter, we'll take up the second half of XSL: the formatting objects vocabulary. Formatting objects are an extremely powerful way of specifying the precise layout you want your pages to have. XSL transformations are used to transform an XML document into an XSL formatting object document.



