2nd Edition

Covers XML Schema

XML IN A NUTSHELL

A Desktop Quick Reference

O'REILLY®

Elliotte Rusty Harold & W. Scott Means

XML

IN A NUTSHELL

Second Edition

Elliotte Rusty Harold
& W. Scott Means

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

9
XPath

XPath is a non-XML language for identifying particular parts of XML documents. XPath lets you write expressions that refer to the first `person` element in a document, the seventh child element of the third `person` element, the `ID` attribute of the first `person` element whose contents are the string “Fred Jones”, all `xml-stylesheet` processing instructions in the document’s prolog, and so forth. XPath indicates nodes by position, relative position, type, content, and several other criteria. XSLT uses XPath expressions to match and select particular elements in the input document for copying into the output document or further processing. XPointer uses XPath expressions to identify the particular point in or part of an XML document to which an XLink links. The W3C XML Schema Language uses XPath expressions to define uniqueness and co-occurrence constraints. XForms relies on XPath to bind form controls to instance data, express constraints on user-entered values, and calculate values that depend on other values.

XPath expressions can also represent numbers, strings, or Booleans. This lets XSLT stylesheets carry out simple arithmetic for purposes such as numbering and cross-referencing figures, tables, and equations. String manipulation in XPath lets XSLT perform tasks such as making the title of a chapter uppercase in a headline or extracting the last two digits from a year.

The Tree Structure of an XML Document

An XML document is a tree made up of nodes. Some nodes contain one or more other nodes. There is exactly one root node, which ultimately contains all other nodes. XPath is a language for picking nodes and sets of nodes out of this tree. From the perspective of XPath, there are seven kinds of nodes:

- The root node
- Element nodes
- Text nodes

- Attribute nodes
- Comment nodes
- Processing-instruction nodes
- Namespace nodes

One thing to note are the constructs not included in this list: CDATA sections, entity references, and document type declarations. XPath operates on an XML document after all these items have been merged into the document. For instance, XPath cannot identify the first CDATA section in a document or tell whether a particular attribute value was directly included in the source element start-tag or merely defaulted from the declaration of the element in a DTD.

Consider the document in Example 9-1. This exhibits all seven kinds of nodes. Figure 9-1 is a diagram of the tree structure of this document.

Example 9-1: The example XML document used in this chapter

```
<?xml version="1.0"?>
<?xml-stylesheet type="application/xml" href="people.xsl"?>
<!DOCTYPE people [
 <!ATTLIST homepage xlink:type CDATA #FIXED "simple"
                  xmlns:xlink CDATA #FIXED "http://www.w3.org/1999/xlink">
 <!ATTLIST person id ID #IMPLIED>
]>
<people>

  <person born="1912" died="1954" id="p342">
    <name>
      <first_name>Alan</first_name>
      <last_name>Turing</last_name>
    </name>
    <!-- Did the word computer scientist exist in Turing's day? -->
    <profession>computer scientist</profession>
    <profession>mathematician</profession>
    <profession>cryptographer</profession>
    <homepage xlink:href="http://www.turing.org.uk/"/>
  </person>

  <person born="1918" died="1988" id="p4567">
    <name>
      <first_name>Richard</first_name>
      <middle_initial>&#x50;</middle_initial>
      <last_name>Feynman</last_name>
    </name>
    <profession>physicist</profession>
    <hobby>Playing the bongoes</hobby>
  </person>

</people>
```

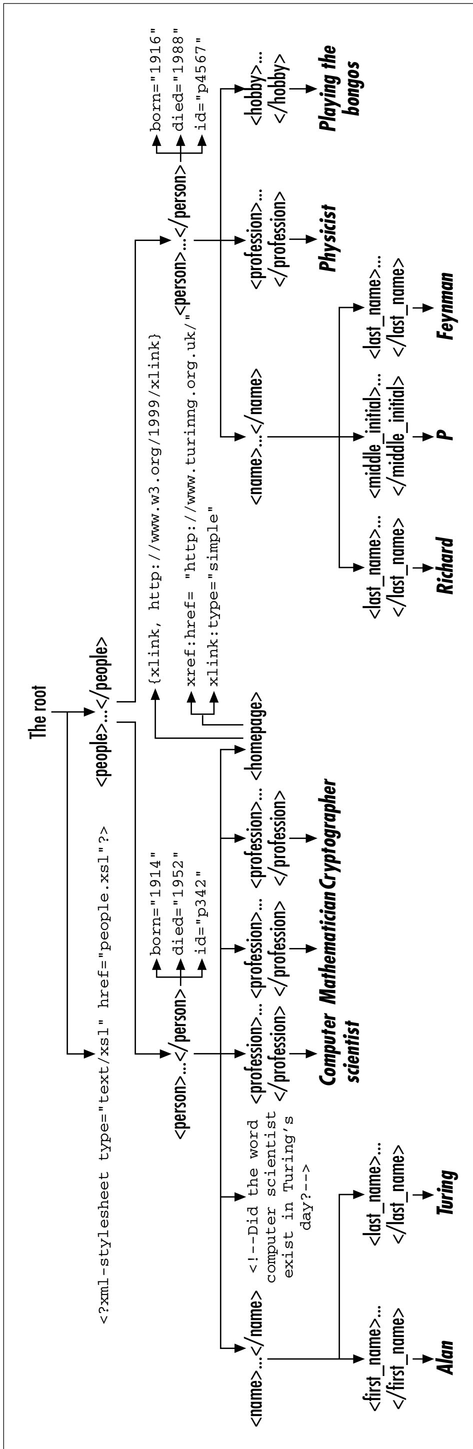


Figure 9-1. The tree structure of Example 9-1

The XPath data model has several nonobvious features. First of all, the root node of the tree is *not* the same as the root element. The root node of the tree contains the entire document including the root element, as well as any comments and processing instructions that occur before the root element start-tag or after the root element end-tag. In Example 9-1, this means the root node contains the `xml-stylesheet` processing instruction, as well as the root element `people`.

However, the XPath data model does not include everything in the document. In particular, the XML declaration, the DOCTYPE declaration, and the various parts of the DTD are *not* addressable via XPath, though if the DTD provides default values for any attributes, then those attributes are noted by XPath. The `homepage` element has an `xlink:type` attribute that was supplied by the DTD. Similarly, any references to parsed entities are resolved. Entity references, character references, and CDATA sections are not individually identifiable, though any data they contain is addressable. For example, XSLT cannot make all the text in CDATA sections bold because XPath doesn't know which text is and isn't part of a CDATA section.

Finally, `xmlns` and `xmlns:`*`prefix`* attributes are not considered attribute nodes, even though that's how a non-namespace-aware parser will see them. However, namespace nodes are attached to every element and attribute node for which a declaration has scope. They are not only attached to the single element where the namespace is declared.

Location Paths

The most useful XPath expression is a *location path*. A location path identifies a set of nodes in a document. This set may be empty, may contain a single node, or may contain several nodes. These can be element nodes, attribute nodes, namespace nodes, text nodes, comment nodes, processing instruction nodes, root nodes, or any combination of these. A location path is built out of successive *location steps*. Each location step is evaluated relative to a particular node in the document called the *context node*.

The Root Location Path

The simplest location path is the one that selects the root node of the document. This is simply the forward slash (`/`). (You'll notice that a lot of XPath syntax is deliberately similar to the syntax used by the Unix shell. Here `/` is the root node of a Unix filesystem, and `/` is the root node of an XML document.) For example, this XSLT template rule uses the XPath pattern `/` to match the entire input document tree and wrap it in an `html` element:

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

`/` is an absolute location path because no matter what the context node is—that is, no matter where you were in the input document when this template rule was applied—it always means the same thing: the root node of the document. It is relative to which document you're processing, but not to anything within that document.

Child Element Location Steps

The second simplest location path is a single element name. This path selects all child elements of the context node with the specified name. For example, the XPath profession refers to all profession child elements of the context node. Exactly which elements these are depends on what the context node is, so this is a relative XPath. For example, if the context node is the Alan Turing person element in Example 9-1, then the location path profession refers to these three profession child elements of that element:

```
<profession>computer scientist</profession>
<profession>mathematician</profession>
<profession>cryptographer</profession>
```

However, if the context node is the Richard Feynman person element in Example 9-1, then the XPath profession refers to its single profession child element:

```
<profession>physicist</profession>
```

If the context node is the name child element of Richard Feynman or Alan Turing's person element, then this XPath doesn't refer to anything at all because neither of those has any profession child elements.

In XSLT, the context node for an XPath expression used in the select attribute of xsl:apply-templates and similar elements is the node that is currently matched. For example, consider the simple stylesheet in Example 9-2. In particular, look at the template rule for the person element. The XSLT processor will activate this rule twice, once for each person node in the document. The first time the context node is set to Alan Turing's person element. The second time the context node is set to Richard Feynman's person element. When the same template is instantiated with a different context node, the XPath expression in <xsl:value-of select="name"/> refers to a different element, and the output produced is therefore different.

Example 9-2: A very simple stylesheet for Example 9-1

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="people">
    <xsl:apply-templates select="person"/>
  </xsl:template>

  <xsl:template match="person">
    <xsl:value-of select="name"/>
  </xsl:template>

</xsl:stylesheet>
```

When XPath is used in other systems, such as XPointer or XForms, other means are provided for determining what the context node is.

Attribute Location Steps

Attributes are also part of XPath. To select a particular attribute of an element, use an @ sign followed by the name of the attribute you want. For example, the XPath expression @born selects the born attribute of the context node. Example 9-3 is a simple XSLT stylesheet that generates an HTML table of names and birth and death dates from documents like Example 9-1.

Example 9-3: An XSLT stylesheet that uses root, child element, and attribute location steps

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

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

  <xsl:template match="people">
    <table>
      <xsl:apply-templates select="person"/>
    </table>
  </xsl:template>

  <xsl:template match="person">
    <tr>
      <td><xsl:value-of select="name"/></td>
      <td><xsl:value-of select="@born"/></td>
      <td><xsl:value-of select="@died"/></td>
    </tr>
  </xsl:template>

</xsl:stylesheet>
```

The stylesheet in Example 9-3 has three template rules. The first template rule has a match pattern that matches the root node, /. The XSLT processor activates this template rule and sets the context node to the root node. Then it outputs the start-tag <html>. This is followed by an xsl:apply-templates element that selects nodes matching the XPath expression people. If the input document is Example 9-1, then there is exactly one such node, the root element. This is selected and its template rule, the one with the match pattern of people, is applied. The XSLT processor sets the context node to the root people element and then begins processing the people template. It outputs a <table> start-tag and then encounters an xsl:apply-templates element that selects nodes matching the XPath expression person. Two child elements of this context node match the XPath expression person so they're each processed in turn using the person template rule. When it begins processing each person element, the XSLT

processor sets the context node to that element. It outputs that element's `name` child element value and `born` and `died` attribute values wrapped in a table row and three table cells. The net result is:

```
<html>
   <table>
      <tr>
         <td>
            Alan
            Turing

         </td>
         <td>1912</td>
         <td>1954</td>
      </tr>
      <tr>
         <td>
            Richard
            P
            Feynman

         </td>
         <td>1918</td>
         <td>1988</td>
      </tr>
   </table>
</html>
```

The comment(), text(), and processing-instruction( ) Location Steps

Although element, attribute, and root nodes account for 90% or more of what you need to do with XML documents, this still leaves four kinds of nodes that need to be addressed: namespace nodes, text nodes, processing-instruction nodes, and comment nodes. Namespace nodes are rarely handled explicitly. The other three node types have special node tests to match them. These are as follows:

- `comment( )`
- `text( )`
- `processing-instruction( )`

Since comments and text nodes don't have names, the `comment( )` and `text( )` node tests match any comment or text node in the context node. Each comment is a separate comment node. Each text node contains the maximum possible contiguous run of text not interrupted by any tag. Entity references and CDATA sections are resolved into text and markup and do not interrupt text nodes.

By default, XSLT stylesheets do process text nodes but do not process comment nodes. You can add a comment template rule to an XSLT stylesheet so it will process comments too. For example, this template rule replaces each comment with the text "Comment Deleted" in italic:

```
<xsl:template match="comment( )">
  <i>Comment Deleted</i>
</xsl:template>
```

With no arguments, the `processing-instruction( )` node test selects all processing-instruction children of the context node. If it has an argument, then it only selects the processing-instruction children with the specified target. For example, the XPath expression `processing-instruction('xml-stylesheet')` selects all processing-instruction children of the context node whose target is `xml-stylesheet`.

Wildcards

Wildcards match different element and node types at the same time. There are three of these: `*`, `node( )`, and `@*`.

The asterisk (*) matches any element node regardless of name. For example, this XSLT template rule says that all elements should have their child elements processed but should not result in any output in and of themselves:

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

The * does not match attributes, text nodes, comments, or processing-instruction nodes. Thus, in the previous example output will only come from child elements that have their own template rules that override this one.

You can put a namespace prefix in front of the asterisk. In this case, only elements in the same namespace are matched. For example, `svg:*` matches all elements with the same namespace URI as the `svg` prefix is mapped to. As usual, it's the URI that matters, not the prefix. The prefix can be different in the stylesheet and the source document as long as the namespace URI is the same.

The `node( )` wildcard matches not only all element types but also text nodes, processing-instruction nodes, namespace nodes, attribute nodes, and comment nodes.

The `@*` wildcard matches all attribute nodes. For example, this XSLT template rule copies the values of all attributes of a `person` element in the document into the content of an `attributes` element in the output:

```
<xsl:template match="person">
  <attributes><xsl:apply-templates select="@*"/></attributes>
</xsl:template>
```

As with elements, you can attach a namespace prefix to the wildcard only to match attributes in a specific namespace. For instance, `@xlink:*` matches all XLink attributes provided that the prefix `xlink` is mapped to the `http://www.w3.org/1999/xlink` URI. Again, it's the URI that matters, not the actual prefix.

Multiple Matches with |

You often want to match more than one type of element or attribute but not all types. For example, you may want an XSLT template that applies to the `profession` and `hobby` elements but not to the `name`, `person`, or `people` elements. You can combine location paths and steps with the vertical bar (|) to indicate that you want to match any of the named elements. For instance, `profession|hobby` matches `profession` and `hobby` elements. `first_name | middle_initial | last_name` matches `first_name`, `middle_initial`, and `last_name` elements. `@id|@xlink:type`

matches `id` and `xlink:type` attributes. `*|@*` matches elements and attributes but does not match text nodes, comment nodes, or processing instruction nodes. For example, this XSLT template rule applies to all the nonempty leaf elements (elements that don't contain any other elements) of Example 9-1:

```
<xsl:template match="first_name|last_name|profession|hobby">
  <xsl:value-of select="text( )"/>
</xsl:template>
```

Compound Location Paths

The XPath expressions you've seen so far—element names, `@` plus an attribute name, `/`, `comment( )`, `text( )`, and `processing-instruction( )`—are all single location steps. You can combine these with the forward slash to move around the hierarchy from the matched node to other nodes. Furthermore, you can use a period to refer to the context node, a double period to refer to the parent node, and a double forward slash to refer to descendants of the context node. With the exception of `//`, these are all similar to Unix shell syntax for navigating a hierarchical filesystem.

Building Compound Location Paths from Location Steps with /

Location steps can be combined with a forward slash (`/`) to make a compound location path. Each step in the path is relative to the one that preceded it. If the path begins with `/`, then the first step in the path is relative to the root node. Otherwise, it's relative to the context node. For example, consider the XPath expression `/people/person/name/first_name`. This begins at the root node, then selects all `people` element children of the root node, then all `person` element children of those nodes, then all `name` children of those nodes, and finally all `first_name` children of those nodes. Applied to Example 9-1, it indicates these two elements:

```
<first_name>Alan</first_name>
<first_name>Richard</first_name>
```

To indicate only the textual content of those two nodes, we have to go one step further. The XPath expression `/people/person/name/first_name/text( )` selects the strings "Alan" and "Richard" from Example 9-1.

These two XPath expressions both began with `/`, so they're absolute location paths that start at the root. Relative location paths can also count down from the context node. For example, the XPath expression `person/@id` selects the `id` attribute of the `person` child elements of the context node.

Selecting from Descendants with //

A double forward slash (`//`) selects from all descendants of the context node, as well as the context node itself. At the beginning of an XPath expression, it selects from all descendants of the root node. For example, the XPath expression `//name` selects all `name` elements in the document. The expression `//@id` selects all the `id`

attributes of any element in the document. The expression person//@id selects all the id attributes of any element contained in the person child elements of the context node, as well as the id attributes of the person elements themselves.

Selecting the Parent Element with ..

A double period (..) indicates the parent of the current node. For example, the XPath expression //@id identifies all id attributes in the document. Therefore, //@id/.. identifies all elements in the document that have id attributes. The XPath expression //middle_initial/../first_name identifies all first_name elements that are siblings of middle_initial elements in the document. Applied to Example 9-1, this selects <first_name>Richard</first_name> but not <first_name>Alan</first_name>.

Selecting the Context Node with .

Finally, the single period (.) indicates the context node. In XSLT this is most commonly used when you need to take the value of the currently matched node. For example, this template rule copies the content of each comment in the input document to a span element in the output document:

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

The . given as the value of the select attribute of xsl:value-of stands for the matched node. This works equally well for element nodes, attribute nodes, and all the other kinds of nodes. For example, this template rule matches name elements from the input document and copies their value into strongly emphasized text in the output document:

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

Predicates

In general, an XPath expression may refer to more than one node. Sometimes this is what you want, but sometimes you want to further winnow the node-set. You want to select only some of the nodes the expression returns. Each step in a location path may (but does not have to) have a predicate that selects from the node list current at that step in the expression. The predicate contains a Boolean expression, which is tested for each node in the context node list. If the expression is false, then that node is deleted from the list. Otherwise, it's retained.

For example, suppose you want to find all profession elements whose value is "physicist." The XPath expression //profession[. = "physicist"] does this. Here the period stands for the string value of the current node, the same as would be returned by xsl:value-of. You can use single quotes around the string instead of double quotes, which is often useful when the XPath expression appears inside a double-quoted attribute value, for example, <xsl:apply-templates select="//profession[.= 'physicist']" />.

If you want to ask for all person elements that have a profession child element with the value "physicist," you'd use the XPath expression //person[profession="physicist"]. If you want to find the person element with id p4567, put an @ in front of the name of the attribute as in //person[@id="p4567"].

As well as the equals sign, XPath supports a full complement of relational operators including <, >, >=, <=, and !=. For instance, the expression //person[@born<=1976] locates all person elements in the document with a born attribute whose numeric value is less than or equal to 1976. Note that if this expression is used inside an XML document, you still have to escape the less-than sign as <, for example, <xsl:apply-templates select="//person[@born <= 1976]"/>. XPath doesn't get any special exemptions from the normal well-formedness rules of XML. On the other hand, if the XPath expression appears outside of an XML document, as it may in some uses of XPointer, then you may not need to escape the less-than sign.

XPath also provides Boolean and and or operators to combine expressions logically. For example, the XPath expression //person[@born<=1920 and @born>=1910] selects all person elements with born attribute values between 1910 and 1920 inclusive. //name[first_name="Richard" or first_name="Dick"] selects all name elements that have a first_name child with the value of either Richard or Dick.

In some cases the predicate may not be a Boolean, but it can be converted to one in a straightforward fashion. Predicates that evaluate to numbers are true if they're equal to the position of the context node, otherwise false. Predicates that indicate node-sets are true if the node-set is nonempty and false if it's empty. String values are true if the string isn't the empty string, false if it is. For example, suppose you want to select only those name elements in the document that have a middle_initial child element. The XPath expression //name selects all name elements. The XPath expression //name[middle_initial] selects all name elements and then checks each one to see if it has a middle_initial child element. Only those that do are retained. When applied to Example 9-1, this expression indicates Richard P. Feynman's name element but not Alan Turing's.

Any or all of the location steps in a location path can have predicates. For example, the XPath expression /people/person[@born < 1950]/name[first_name = "Alan"] first selects all people child elements of the root element (of which there's exactly one in Example 9-1). Then from those it chooses all person elements whose born attribute has a value numerically less than 1950. Finally, from that group of elements, it selects all name child elements that have a first_name child element with the value "Alan."

Unabbreviated Location Paths

Up until this point, we've been using what are called *abbreviated location paths*. These are much easier to type, much less verbose, and much more familiar to most people. They're also the kind of XPath expression that works best for XSLT match patterns. However, XPath also offers an unabbreviated syntax for location paths, which is more verbose but perhaps less cryptic and definitely more flexible.

Every location step in a location path has two required parts, an axis and a node test, and one optional part, the predicates. The axis tells you which direction to travel from the context node to look for the next nodes. The node test tells you which nodes to include along that axis, and the predicates further reduce the nodes according to some expression.

In an abbreviated location path, the axis and the node test are combined, while in an unabbreviated location path, they're separated by a double colon (`::`). For example, the abbreviated location path `people/person/@id` is composed of three location steps. The first step selects `people` element nodes along the child axis. The second step selects `person` element nodes along the child axis. The third step selects `id` attribute nodes along the attribute axis. When rewritten using the unabbreviated syntax, the same location path is `child::people/child::person/attribute::id`.

These full, unabbreviated location paths may be absolute if they start from the root node, just as abbreviated paths can be. For example, the full form `/child::people/child::person` is equivalent to the abbreviated form `/people/person`.

Unabbreviated location paths may have and be used in predicates as well. For example, the abbreviated path `/people/person[@born<1950]/name[first_name="Alan"]` becomes `/child::people/child::person[ attribute::born < 1950 ] /child::name[ child::first_name = "Alan" ]` in the full form.

Overall, the unabbreviated form is quite verbose and not much used in practice. It isn't even allowed in XSLT match patterns. However, it does offer one crucial ability that makes it essential to know: it is the only way to access most of the axes from which XPath expressions can choose nodes. The abbreviated syntax lets you walk along the child, parent, self, attribute, and descendant-or-self axes. The unabbreviated syntax adds eight more:

The ancestor axis
: All element nodes that contain the context node, that is, the parent node, the parent's parent, the parent's parent's parent, and so on up through the root node in reverse document order.

The following-sibling axis
: All nodes that follow the context node and are children of the same parent node in document order. Attribute and namespace nodes do not have any siblings.

The preceding-sibling axis
: All nodes that precede the context node and are children of the same parent node in reverse document order. Attribute and namespace nodes do not have any siblings.

The following axis
: All nodes that follow the end of the context node in document order except for attribute and namespace nodes.

The preceding axis
: All nodes that precede the start of the context node in reverse document order except for attribute and namespace nodes.

The namespace axis
: All namespaces in scope on the context node, whether declared on the context node or one of its ancestors.

The descendant axis
: All descendants of the context node but not the context node itself.

The ancestor-or-self axis
: All ancestors of the context node and the context node itself.

Example 9-4 demonstrates several of these axes using the full unabbreviated syntax. The goal is to produce a list of `person` elements that look more or less like this (after accounting for whitespace):

```
<dt>Richard P Feynman</dt>
<dd>
  <ul>
    <li>physicist</li>
    <li>Playing the bongoes</li>
  </ul>
</dd>
```

Example 9-4: An XSLT stylesheet that uses unabbreviated XPath syntax

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <dl>
      <xsl:apply-templates select="descendant::person"/>
    </dl>
  </xsl:template>

  <xsl:template match="person">
    <dt><xsl:value-of select="child::name"/></dt>
    <dd>
      <ul>
        <xsl:apply-templates select="child::name/following-sibling::*"/>
      </ul>
    </dd>
  </xsl:template>

  <xsl:template match="*">
    <li><xsl:value-of select="self::*"/></li>
  </xsl:template>

  <xsl:template match="homepage"
               xmlns:xlink="http://www.w3.org/1999/xlink">
    <li><xsl:value-of select="attribute::xlink:href"/></li>
  </xsl:template>

</xsl:stylesheet>
```

The first template rule matches the root node. It applies templates to all descendants of the root node that happen to be person elements. That is, it moves from the root node along the descendant axis with a node test of person. This XPath expression could have been rewritten in the abbreviated syntax as //person.

The second template rule matches person elements. It places the value of the name child of each person element in a dt element. The location path used here, child::name, could have been rewritten in the abbreviated syntax as the single word name. Then it applies templates to all elements that follow the name element at the same level of the hierarchy. It begins at the context node person element, then moves along the child axis to find the name element. From there it moves along the following-sibling axis looking for elements of any type (*) after the name element that are also children of the same person element. There is no abbreviated equivalent for the following-sibling axis, so this really is the simplest way to make this statement.

The third template rule matches any element not matched by another template rule. It simply wraps that element in an li element. The XPath self::* selects the value of the currently matched element, that is, the context node. This expression could have been abbreviated as a single period.

The fourth and final template rule matches homepage elements. In this case we need to select the value of xlink:href attribute, so we move from the context homepage node along the attribute axis. The node test is looking for the xlink:href attributes. (More properly, it's looking for an attribute with the local name href whose prefix is mapped to the *http://www.w3.org/1999/xlink* namespace URI.)

General XPath Expressions

So far we've focused on the very useful subset of XPath expressions called *location paths*. Location paths identify a set of nodes in an XML document and are used in XSLT match patterns and select expressions. However, location paths are not the only possible type of XPath expression. XPath expressions can also return numbers, Booleans, and strings. For instance, these are all legal XPath expressions:

- `3.141529`
- `2+2`
- `'Rosalind Franklin'`
- `true()`
- `32.5 < 76.2`
- `position()=last()`

XPath expressions that aren't node-sets can't be used in the match attribute of an xsl:template element. However, they can be used as values for the select attribute of xsl:value-of elements, as the well as in location path predicates.

Numbers

There are no pure integers in XPath. All numbers are 8-byte, IEEE 754 floating-point doubles, even if they don't have an explicit decimal point. This format is identical to Java's `double` primitive type. As well as representing floating-point numbers ranging from 4.94065645841246544e-324 to 1.79769313486231570e+308 (positive or negative) and zero, this type includes special representations of positive and negative infinity and a special not a number value (NaN) used as the result of operations like dividing zero by zero.

XPath provides the five basic arithmetic operators that will be familiar to any programmer:

+
: Addition

-
: Subtraction

*
: Multiplication

div
: Division

mod
: Taking the remainder

The more common forward slash couldn't be used for division because it's already used to separate location steps in a location path. Consequently, a new operator had to be chosen. The word `mod` was chosen instead of the more common `%` operator. Aside from these minor differences in syntax, all five operators behave exactly as they do in Java. For instance, 2+2 is 4, 6.5 div 1.5 is 4.33333333, 6.5 mod 1.5 is 0.5, and so on. Placing the element <xsl:value-of select="6*7"/> in an XSLT template inserts the string `42` into the output tree when the template is instantiated. More often, a stylesheet performs some simple arithmetic on numbers read from the input document. For instance, this template rule calculates the century in which a person was born:

```
<xsl:template match="person">
  <century>
    <xsl:value-of select="(@born - (@born mod 100)) div 100"/>th
  </century>
</xsl:template>
```

Strings

XPath strings are ordered sequences of Unicode characters such as "Fred", "Ethel", "   ", or "Ξηνος". String literals may be enclosed in either single or double quotes as convenient. The quotes are not themselves part of the string. The only restriction XPath places on a string literal is that it must not contain the kind of quote that delimits it. That is, if the string contains single quotes, it has to be enclosed in double quotes and vice versa. String literals may contain whitespace including tabs, carriage returns, and line feeds, as well as back slashes

and other characters that would be illegal in many programming languages. However, if the XPath expression is part of an XML document, some of these possibilities may be ruled out by XML's well-formedness rules, depending on context.

You can use the = and != comparison operators to check whether two strings are the same. You can also use the relational <, >, <=, and >= operators to compare strings, but unless both strings clearly represent numbers (e.g., "-7.5" or '54.2'), the results are unlikely to make sense. In general, you can't define any real notion of string order in Unicode without detailed knowledge of the language in which the string is written.

Other operations on strings are provided by XPath functions and will be discussed shortly.

Booleans

A Boolean is a value that has exactly two states, true or false. Every Boolean must have one of these binary values. XPath does not provide any Boolean literals. If you use `<xsl:value-of select="true"/>` in an XSLT stylesheet, then the XSLT processor looks for a child element of the context node named true. However, the XPath functions true( ) and false( ) can substitute for the missing literals quite easily.

Most of the time, however, Booleans are created by comparisons between other objects, most commonly numbers. XPath provides all the usual relational operators including =, !=, <, >, >=, and <=. In addition, the and and or operators can combine Boolean expressions according to the usual rules of logic.

Booleans are most commonly used in predicates of location paths. For example, in the location step person[profession="physicist"], profession="physicist" is a Boolean. It is either true or false; there is no other possibility. Booleans are also commonly used in the test attribute of xsl:if and xsl:when elements. For example, this XSLT template rule includes the profession element in the output only if its contents are "physicist" or "computer scientist":

```
<xsl:template match="profession">
  <xsl:if test=".='computer scientist' or .='physicist'">
    <xsl:value-of select="."/>
  </xsl:if>
</xsl:template>
```

This XSLT template rule italicizes the profession element if and only if its content is the string "computer scientist":

```
<xsl:template match="profession">
  <xsl:choose>
    <xsl:when test=".='computer scientist'">
      <i><xsl:value-of select="."/></i>
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="."/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

Finally, there's a `not( )` function that reverses the sense of its Boolean argument. For example, if `.='computer scientist'` is true, then `not(.='computer scientist')` is false and vice versa.

XPath Functions

XPath provides a number of functions that you may find useful in predicates or raw expressions. All of these are discussed in Chapter 22. For example, the `position( )` function returns the position of the current node in the context node list as a number. This XSLT template rule uses the `position( )` function to calculate the number of the `person` being processed, relative to other nodes in the context node list:

```
<xsl:template match="person">
  Person <xsl:value-of select="position( )"/>,
  <xsl:value-of select="name"/>
</xsl:template>
```

Each XPath function returns one of these four types:

- Boolean
- Number
- Node-set
- String

There are no void functions in XPath. Therefore, XPath is not nearly as strongly typed as languages like Java or even C. You can often use any of these types as a function argument regardless of which type the function expects, and the processor will convert it as best it can. For example, if you insert a Boolean where a string is expected, then the processor will substitute one of the two strings "true" and "false" for the Boolean. The one exception is functions that expect to receive node-sets as arguments. XPath cannot convert strings, Booleans, or numbers to node-sets.

Functions are identified by the parentheses at the end of the function names. Sometimes these functions take arguments between the parentheses. For instance, the `round( )` function takes a single number as an argument. It returns the number rounded to the nearest integer. For example, `<xsl:value-of select="round(3.14)"/>` inserts `3` into the output tree.

Other functions take more than one argument. For instance, the `starts-with( )` function takes two arguments, both strings. It returns true if the first string starts with the second string. For example, this XSLT `apply-templates` element selects all `name` elements whose last name begins with T:

```
<xsl:apply-templates select="name[starts-with(last_name, 'T')]"/>
```

In this example the first argument to the `starts-with( )` function is actually a node-set, not a string. The XPath processor converts that node-set to its string value (the text content of the first element in that node-set) before checking to see whether it starts with T.

Some XSLT functions have variable-length argument lists. For instance, the concat( ) function takes as arguments any number of strings and returns one string formed by concatenating all those strings together in order. For example, concat("a", "b", "c", "d") returns "abcd".

In addition to the functions defined in XPath and discussed in this chapter, most uses of XPath, such as XSLT and XPointer, define many more functions that are useful in their particular context. You use these extra functions just like the built-in functions when you're using those applications. XSLT even lets you write extension functions in Java and other languages that can do almost anything, for example, making SQL queries against a remote database server and returning the result of the query as a node-set.

Node-Set Functions

The node-set functions either operate on or return information about node-sets, that is, collections of XPath nodes. You've already encountered the position( ) function. Two related functions are last( ) and count( ). The last( ) function returns the number of nodes in the context node list, which also happens to be the same as the position of the last node in the list. The count( ) function is similar except that it returns the number of nodes in its node-set argument rather than in the context node list. For example, count(//name) returns the number of name elements in the document. Example 9-5 uses the position( ) and count( ) functions to list the people in the document in the form "Person 1 of 10, Person 2 of 10, Person 3 of 10…" In the second template the position( ) function determines which person element is currently being processed, and the count( ) function determines how many total person elements there are in the document.

Example 9-5: An XSLT stylesheet that uses the position() and count() functions

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="people">
    <xsl:apply-templates select="person"/>
  </xsl:template>

  <xsl:template match="person">
    Person <xsl:value-of select="position( )"/>
    of <xsl:value-of select="count(//person)"/>:
    <xsl:value-of select="name"/>
  </xsl:template>

</xsl:stylesheet>
```

The id( ) function takes as an argument a string containing one or more IDs separated by whitespace and returns a node-set containing all the nodes in the document that have those IDs. These are attributes declared to have type ID in the DTD not necessarily attributes named ID or id. (A DTD must be both present and

processed by the parser for the id() function to work.) Thus, in Example 9-1, id('p342') indicates Alan Turing's person element; id('p342 p4567') indicates both Alan Turing and Richard Feynman's person elements.

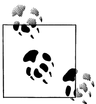
The id( ) function is most commonly used in the abbreviated XPath syntax. It allows you to form absolute location paths that don't start from the root. For example, id('p342')/name refers to Alan Turing's name element, regardless of where Alan Turing's person element is in the document, as long as it hasn't changed ID. This function is especially useful for XPointers where it takes the place of HTML's named anchors.

Finally, there are three node-set functions related to namespaces. The local-name( ) function takes as an argument a node-set and returns the local part of the first node in that set. The namespace-uri( ) function takes a node-set as an argument and returns the namespace URI of the first node in the set. Finally, the name( ) function takes a node-set as an argument and returns the prefixed name of the first node in that set. In all three functions the argument may be omitted, in which case the context node's namespace is evaluated. For instance, when applied to Example 9-1 the XPath expression, local-name(//homepage/@xlink:href) is href; namespace-uri(//homepage/@xlink:href) is http://www.w3.org/1999/xlink; and name(//homepage/@xlink:href) is xlink:href.

String Functions

XPath includes functions for basic string operations such as finding the length of a string or changing letters from upper- to lowercase. It doesn't have the full power of the string libraries in Python or Perl—for instance, there's no regular expression support—but it's sufficient for many simple manipulations you need for XSLT or XPointer.

The string( ) function converts an argument of any type to a string in a reasonable fashion. Booleans are converted to the string "true" or the string "false." Node-sets are converted to the string value of the first node in the set. This is the same value calculated by the xsl:value-of element. That is, the string value of the element is the complete text of the element after all entity references are resolved and tags, comments, and processing instructions have been stripped out. Numbers are converted to strings in the format used by most programming languages, such as "1987," "299792500," or "2.71828."



In XSLT the xsl:decimal-format element and format-number( ) function provide more precise control over formatting so you can insert separators between groups, change the decimal separator, use non-European digits, and make similar adjustments.

The normal use of most of the rest of the string functions is to manipulate or address the text content of XML elements or attributes. For instance, if date attributes were given in the format MM/DD/YYYY, then the string functions would allow you to target the month, day, and year separately.

The `starts-with( )` function takes two string arguments. It returns true if the first argument starts with the second argument. For example, `starts-with('Richard', 'Ric')` is true but `starts-with('Richard', 'Rick')` is false. There is no corresponding `ends-with( )` function.

The `contains( )` function also takes two string arguments. However, it returns true if the first argument contains the second argument—that is, if the second argument is a substring of the first argument—regardless of position. For example, `contains('Richard', 'ar')` is true but `contains('Richard', 'art')` is false.

The `substring-before( )` function takes two string arguments and returns the substring of the first argument that precedes the initial appearance of the second argument. If the second string doesn't appear in the first string, then `substring-before( )` returns the empty string. For example, `substring-before('MM/DD/YYYY', '/')` is `MM`. The `substring-after( )` function also takes two string arguments but returns the substring of the first argument that follows the initial appearance of the second argument. If the second string doesn't appear in the first string, then `substring-after( )` returns the empty string. For example, `substring-after ('MM/DD/YYYY', '/')` is `'DD/YYYY'`. `substring-before(substring-after('MM/DD/YYYY', '/')', '/')` is `DD`. `substring-after(substring-after('MM/DD/YYYY', '/')', '/')` is `YYYY`.

If you know the position of the substring you want, then you can use the `substring( )` method instead. This takes three arguments: the string from which the substring will be copied, the position in the string from which to start extracting, and the number of characters to copy to the substring. The third argument may be omitted, in which case the substring contains all characters from the specified start position to the end of the string. For example, `substring('MM/DD/YYYY', 1, 2)` is `MM`; `substring('MM/DD/YYYY', 4, 2)` is `DD`; and `substring('MM/DD/YYYY', 7)` is `YYYY`.

The `string-length( )` function returns a number giving the length of its argument's string value or the context node if no argument is included. In Example 9-1, `string-length(//name[position( )=1])` is 29. If that seems long to you, remember that all whitespace characters are included in the count. If it seems short to you, remember that markup characters are not included in the count.

Theoretically, you could use these functions to trim and normalize whitespace in element content. However, since this would be relatively complex and is such a common need, XPath provides the `normalize-space( )` function to do this. For instance, in Example 9-1 the value of `string(//name[position( )=1])` is:

```
Alan
Turing
```

This contains a lot of extra whitespace that was inserted purely to make the XML document neater. However, `normalize-space(string(//name[position( )=1]))` is the much more reasonable:

```
Alan Turing
```

Although a more powerful string-manipulation library would be useful, XSLT is really designed for transforming the element structure of an XML document. It's not meant to have the more general power of a language like Perl, which can handle arbitrarily complicated and varying string formats.

Boolean Functions

The Boolean functions are few in number and quite straightforward. They all return a Boolean that has the value true or false. The `true()` function always returns true. The `false()` function always returns false. These substitute for Boolean literals in XPath.

The `not()` function reverses the sense of its Boolean argument. For example, `not(@id>400)` is almost always equivalent to `(@id<=400)`. (NaN is a special case.)

The `boolean()` function converts its single argument to a Boolean and returns the result. If the argument is omitted, then it converts the context node. Numbers are converted to false if they're zero or NaN. All other numbers are true. Node-sets are false if they're empty, true if they have at least one element. Strings are false if they have zero length, otherwise they're true. Note that according to this rule, the string `"false"` is in fact true.

Number Functions

XPath includes a few simple numeric functions for summing groups of numbers and finding the nearest integer to a number. It doesn't have the full power of the math libraries in Java or Fortran—for instance, there's no square root or exponentiation function—but it's got enough to do most of the basic math you need for XSLT or the even simpler requirements of XPointer.

The `number()` function can take any type as an argument and convert it to a number. If the argument is omitted, then it converts the context node. Booleans are converted to 1 if true and 0 if false. Strings are converted in a plausible fashion. For instance the string "7.5" will be converted to the number 7.5. The string "Fred" will be converted to NaN. Node-sets are converted to numbers by first converting them to their string values and then converting the resulting string to a number. The detailed rules are a little more complex, but as long as the object you're converting can reasonably be interpreted as a single number, chances are the `number()` function will do what you expect. If the object you're converting can't be reasonably interpreted as a single number, then the `number()` function will return NaN.

The `round()`, `floor()`, and `ceiling()` functions all take a single number as an argument. The `floor()` function returns the greatest integer less than or equal to its argument. The `ceiling()` function returns the smallest integer greater than or equal to its argument. The `round()` function returns its argument rounded to the nearest integer. When rounding numbers like 1.5 and –3.5 that are equally close to two integers, `round()` returns the greater of the two possibilities. (This means that –1.5 rounds to –1, but 1.5 rounds to 2.)

The `sum()` function takes a node-set as an argument. It converts each node in the set to its string value, then converts each of those strings to a number. It then adds up the numbers and returns the result.