XPath

Source http://www.oreilly.com/catalog/xmlnut/chapter/ch09.html

XPath is a non-XML language used to identify particular parts of XML documents. XPath lets you write expressions that refer to the document's first person element, 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 that an XLink links to.

XPath expressions can also represent numbers, strings, or Booleans, so XSLT stylesheets carry out simple arithmetic for numbering and cross-referencing figures, tables, and equations. String manipulation in XPath lets XSLT perform tasks like making the title of a chapter uppercase in a headline, but mixed case in a reference in the body text.

The Tree Structure of an XML Document

An XML document is a tree made up of nodes. Some nodes contain other nodes. One root node 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**

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

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

Example 9-1: The Example XML Document Used in This Chapter

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" 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>
```

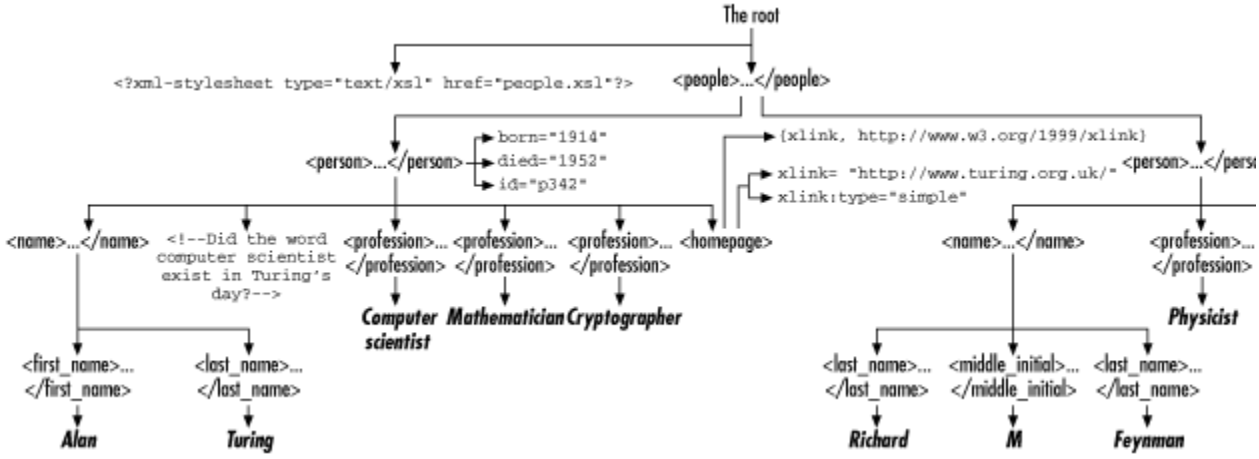
```
]>
<people>

  <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>&#x4D;</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 inobvious features. First, the tree's root node is *not* the same as its root element. The tree's root node contains the entire document, including the root element and comments and processing instructions that occur before the root element start tag or after the root element end tag. In Example 9-1, the root node contains the xml-stylesheet processing instruction and the root element people.

The XPath data model does not include everything in the document. In particular, the XML declaration and DTD are *not* addressable via XPath. However, if the DTD provides default values for any attributes, then XPath recognizes those attributes. The homepage element has an xlink:type attribute 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 does not enable you to make all text in CDATA sections bold because XPath doesn't know what text is and isn't part of a CDATA section.

Finally, xmlns attributes are reported as namespace nodes. They are not considered attribute nodes, though a non-namespace aware parser will see them as such. Furthermore these nodes are attached to every element and attribute node for which that declaration has scope. They are not just attached to the single element where the namespace is declared.

Location Paths

The most useful XPath expression is a *location path*. A location path uses at least one location step to identify a set of nodes in a document. This set may be empty, contain a single node, or contain several nodes. These nodes can be element, attribute, namespace, text, comment, processing instruction, root nodes, or any combination of them.

The Root Location Path

The simplest location path is the one that selects the document's root node. This path is simply the forward slash /. (You'll notice that a lot of XPath syntax was deliberately chosen to be similar to the syntax used by the Unix shell. Here / is the root of a Unix filesystem and / is the root node of an XML document.) For example, this XSLT template 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>
```

The forward slash / is an absolute location path because no matter what the context node is, no matter where you were in the input document when this template was applied, it always means the same thing: the root node of the document. It is relative to the document you process, but not to anything within that document.

Child Element Location Steps

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

```
<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 these elements has 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 currently matched node. Consider the simple stylesheet in Example 9-2. Look at the template for the person element. The XSLT processor activates this template twice, once for each person node in the document. The first time this template rule is activated, the context node is set to Alan Turing's person element. The second time this template rule is activated, the context node is set to Richard Feynman's person element. When the same template is activated 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 other systems, such as XPointer, use XPath, other means are provided for determining the context node.

Attribute Location Steps

Attributes are also part of XPath. To select a particular attribute of an element, use an at 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 such as Example 9-1.

Example 9-3: An XSLT Stylesheet Using 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 templates. The first template has a match pattern that matches the root node, /. The XSLT processor activates this template 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 exactly one such node, the root element, is present. This element is selected and its template, 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. 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 id attribute value, 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
            M
            Feynman

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

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

Though element, attribute, and root nodes account for at least 90 percent of what you need to do with XML documents, four kinds of nodes still need to be addressed: namespace nodes, text nodes, processing instruction nodes, and comment nodes. Namespace nodes are rarely invoked explicitly. Instead, the stylesheet uses its own xmlns attributes and prefixes, and the processor handles any necessary conversion between input namespace prefixes and stylesheet namespace prefixes (as long as the URIs match). On output, the XSLT processor inserts xmlns attributes as necessary to make namespaces in the output document correct. The other three node types have special functions to match them:

- **comment( )**
- **text( )**
- **processing-instruction( )**

Since comments and text nodes don't have names, the comment( ) and text( ) functions match any comment or text node that's an immediate child of the context node. Each comment is a separate comment node. Each text node contains the maximum possible contiguous run of text not interrupted by a 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 to an XSLT stylesheet so it will process comments, too. This template 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( ) function selects all the context node's processing instruction children. If it has an argument, it selects only 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 allow you to match different element and node types at the same time. There are three wild cards: *, node( ), and @*.

The asterisk * matches any element node, regardless of type. For example, this XSLT template says that all elements should have their child elements processed but should not produce any output:

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

The * does not match attributes, text nodes, comments, or processing instruction nodes. Thus in this example, output comes only from child elements that have their own templates 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, the URI, not the prefix, matters. The prefix may differ in the stylesheet and the source document, as long as the namespace URI is the same.

The node( ) wild card matches all nodes: element nodes, text nodes, attribute nodes, processing instruction nodes, namespace nodes, and comment nodes.

The @* wild card matches all attribute nodes. For example, this XSLT template copies the values of all attributes of a person element in the document to 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 wild card to match only 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 namespace. Again, the URI, not the actual prefix, matters.

Multiple Matches with |

You may 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 individual location 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 child elements of the context node. first_name|middle_initial|last_name matches first_name, middle_initial, and last_name child elements of the context node. @id|@xlink:type matches id and xlink:type attributes of the context node. *|@* matches elements and attributes, but does not match text nodes, comment nodes, or processing instruction nodes. This XSLT template 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( ), node(), and processing-instruction( )--are all single location steps. You can combine these location steps with the forward slash to move down the hierarchy from the matched node to other nodes. You can also use a period to refer to the current 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 file system.

Building Compound Location Paths from Location Steps with /

The forward slash / combines different location steps 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 is relative to the context node. For example, consider the XPath expression /people/person/name/first_name. This expression begins at the root node, selects all people element children of the root node, selects all person element children of those nodes, then all name children of those nodes, and finally, all first_name children of the nodes. Applied to Example 9-1, the expression indicates these two elements:

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

To indicate only textual content of those two nodes, we must 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 element of the context node.

Selecting from All 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.

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. 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 that is contained in the person child element of the context node. 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 expression selects <first_name>Richard</first_name>, but not <first_name>Alan</first_name>.

Selecting the Current Element with .

The single period indicates the current node. In XSLT this is most commonly used when you need to take the value of the currently matched element. For example, this template copies the content of each comment in the input document to an italicized span element in the output document:

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

The ., 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 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 need to winnow the node set further to select only some of the nodes the expression returns. Each step in a location path may have a predicate that selects from the node list that is 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, 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"] finds these elements. Here the period stands for the string value of the current node, the same as the value returned by xsl:value-of. You can use single instead of double quotes around the string, which is often useful when the XPath expression appears inside an already-quoted attribute value; for example, <xsl:template match="//profession[.='physicist']">.

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

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

In some cases the predicate may not be a boolean, but converting it to one can be straightforward. Predicates that evaluate to numbers are true if they're equal to the position of the context node, false otherwise. 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 the 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 Listing 9-1, this expression indicates Richard M. Feynman's name element, but not Alan Turing's.

Any or all 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 it chooses all person elements whose born attribute has a value numerically less than 1950 from those elements. Finally, it selects all name child elements that have a first_name child element with the value Alan from that group of elements.

Unabbreviated Location Paths

Up to this point we've used what are called *abbreviated location paths*. These paths are much easier to type, less verbose, and 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 that 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 winnow the nodes according to an expression.

In an abbreviated location path, the axis and the node test are combined. However, they're separated by a double colon :: in an unabbreviated location path. 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 selects person element nodes along the child axis, and the third selects id 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. The full form /child::people/child::person, for example, is equivalent to the unabbreviated form /people/person. Unabbreviated location paths may also have predicates. For example, the abbreviated path /people/person[@born < 1950]/name[first_name = "Alan"] becomes /child::people/child::person[@born < 1950]/child::name[first_name = "Alan"] in the full form.

Overall the unabbreviated form is verbose and not used much in practice. It isn't even allowed in XSLT match patterns. However, it does offer one crucial ability that makes it essential to know. The unabbreviated form 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 axes:

Ancestor axis

All element nodes that contain the context node; the parent node, the parent's parent, the parent's parent's parent, etc., up through the root node in reverse document order

Following-sibling axis

All nodes that follow the context node and are contained in the same parent element node in document order

Preceding-sibling axis

All nodes that precede the context node and are contained in the same parent element node in reverse document order

Following axis

All nodes that follow the end of the context node in document order

Preceding axis

All nodes that precede the start of the context node in reverse document order

Namespace axis

All namespaces in scope on the context node, whether declared on the context node or one of its ancestors

Descendant axis

All the context node's descendants, not including the context node itself

Ancestor-or-self axis

All the context node's ancestors and the context node itself

Using the full, unabbreviated syntax, Example 9-4 demonstrates several of these axes. The example produces a list of person elements that look more or less like this (after accounting for whitespace):

```
<dt>Richard M 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 matches the root node. It applies templates to all descendants of the root node that are person elements; that is, it moves from the root node along the descendant axis with a node test of person.

The second template 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.) Next, 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. No abbreviated equivalent exists for the following-sibling axis, so this really is the simplest way to make the statement.

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

The fourth and final template matches homepage elements. In this case you need to select the value of xlink:href attribute, so move from the context homepage node along the attribute axis. The node test looks for the xlink:href attributes. Specifically, it looks 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 an XPath expression. XPath expressions can also return numbers, booleans, and strings. For instance, all legal XPath expressions include:

- **3.141529**
- **2+2**
- **'Rosalind Franklin'**
- **true( )**
- **32.5 < 76.2E-21**

- **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 and in location path predicates.

Numbers

No pure integers exist in XPath. All numbers are eight-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. Besides 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, such as dividing zero by zero.

XPath provides the five basic arithmetic operators 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 was chosen. Java's % operator was also replaced by the word mod. 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 would insert the string 42 into the output document when the template was processed. More often, you'll perform simple arithmetic on numbers read from the input document. For instance, this template divides the value of the id attribute by 10 and inserts the result into the output:

```
<xsl:template match="person">
  <xsl:value-of select="@id div 10"/>
</xsl:template>
```

Strings

XPath strings are ordered sequences of Unicode characters like "Fred," "Ethel," "كريم " or "Ξηνο." 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 does not contain the type of quote that delimits it. If the string contains single quotes, it must be enclosed in double quotes. and vice versa. String literals may contain whitespace, including tabs, carriage returns, line feeds, backslashes, 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 will probably not make sense. In general, you can't define a real notion of string order in Unicode without detailed knowledge of the text being ordered.

Other operations on strings are provided by XPath functions and are discussed in the following sections.

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 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.

Booleans, however, are usually created by comparisons between other objects, most commonly numbers. XPath provides all the usual relational operators, including =, !=, <, >, >=, and <=. The and and or operators can also be used to combine boolean expressions according to the usual rules of boolean logic.

Booleans are most commonly used in predicates of location paths. In the location path //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 includes the profession element in the output if and 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 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, a not( ) function 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 many functions you may find useful in predicates or in raw expressions. All of these functions are discussed in Chapter 19, XPath Reference. For example, the position( ) function returns the current node's position in the context node list as a number. This XSLT template 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. Otherwise, XPath is not as strongly typed as languages such as Java or C. You can often use these types as a function argument, regardless of which type the function expects, and the processor will convert it the best it can. If you insert a boolean where a string is expected, 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 function names. Sometimes these functions take arguments between the parentheses. 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 xsl: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 last_name element) before checking to see whether it starts with T.

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

Besides the functions defined in XPath and discussed in this chapter, most uses of XPath like XSLT and XPointer define more functions that are useful in their particular context. You use these extra functions just like built-in functions when you use those applications. XSLT even

lets you write extension functions in Java and other languages that can do almost anything, for example, make SQL queries against a remote database server and return the result of the query as a node set.

Node Set Functions

Node set functions either operate on or return information about node sets, ordered 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 set, which is the same as the position of the last node in the set. 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) tells you how many name elements exist in the document. Example 9-5 uses the position( ) and count( ) functions to list people in the document in the form "Person 1 of 10, Person 2 of 10, Person 3 of 10, etc." In the second template the position( ) function tells you which person element is currently being processed, and the count( ) function tells you how many total person elements exist 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 a string containing one or more IDs separated by whitespace as an argument and returns a node set containing all nodes in the document that have those IDs. These are nodes with attributes declared to have type ID in the DTD, not necessarily nodes with attributes named ID or id. Thus in Example 9-1, id('p342') indicates Alan Turing's person element and 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 in XPointers, where it takes the place of HTML's named anchors.

Finally, three node set functions are related to namespaces. The local-name( ) function takes as an argument a node set, most often containing just a single node, and returns the local name 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 set's first node. 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 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 a string's length or changing letters from uppercase to lowercase. It doesn't have the full power of the string libraries in Python or Perl; for example, there's no regular expression support. However, XPath is sufficient for many simple manipulations you need for XSLT or XPointer.

The string( ) function converts any type of argument 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 value is the same value calculated by the xsl:value-of element; the string value of the element is the complete Unicode 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," "2.71828," or "2.998E+10."

TIP:

In XSLT, the xsl:number element provides 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 remaining string functions is to manipulate or address the text content of XML elements or attributes. 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 string that precedes the second argument's initial appearance. 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 string that follows the second argument's initial appearance. If the second string doesn't appear in the first string, 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. and substring-after(substring-after('MM/DD/YYYY', '/')', '/') is YYYY.

If you know the position of the substring you want in a given string, then use the substring( ) method instead. This method takes three arguments: the string from which the substring is copied, the position in the string from which to start extracting, and the number of characters to copy for 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', 2) is DD; and substring('MM/DD/YYYY', 7) is YYYY.

The string-length( ) function returns a number giving the length of the string value of its argument, or of the context node if no argument is included. In Example 9-1, string-length(//name[position( )=1]) is 29. If that value seems long, remember that all whitespace characters are included in the count. If it seems short, 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 process would be relatively complex and is such a common need, XPath provides the normalize-space( ) function to do this. In Example 9-1 the value of string(//name[position( )=1]) is:

Alan
Turing

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

Alan Turing

Though 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 general power of a language like Perl, which can handle arbitrarily complicated and varying string formats.

Boolean Functions

The boolean functions are straightforward and few in number. They all return a boolean with the value true or false. The true( ) function always returns true and the false( ) function always returns false. These functions 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, it converts the context node. Numbers are converted to false if they're zero or NaN (not a number); all other numbers are true. Node sets are false if they're empty and true if they contain at least one node. Strings are false if they have zero length and true otherwise. Note that according to this rule, the string "false" is in fact true.

Number Functions

XPath includes a few simple numeric functions used 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 basic math you need for XSLT or the 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, 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" is converted to the number 7.5 and the string "8.5E2" is converted to the number 8,500. The string "Fred" is 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 more complex, but as long as the object you convert can be reasonably interpreted as a single number, the number( ) function will probably do what you expect. If the object you convert can't be reasonably interpreted as a single number, then the number( ) function returns 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 larger of the two.

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. Finally, it adds the numbers and returns the result.

~~~ End of Article ~~~