XSLT is concerned with mapping portions of the source tree to corresponding portions of the result tree. Instantiating elements and text in the result tree is fairly simple; for the most part, you simply place the things that belong in the result tree into a template rule's template.

The source tree can be accessed in a number of XSLT elements, always by way of a match or select attribute. A template rule might look like this:

```
<xsl:template match="stuff_from_source_tree">
<h1><xsl:value-of select="other_stuff"/></h1>
</xsl:template>
```

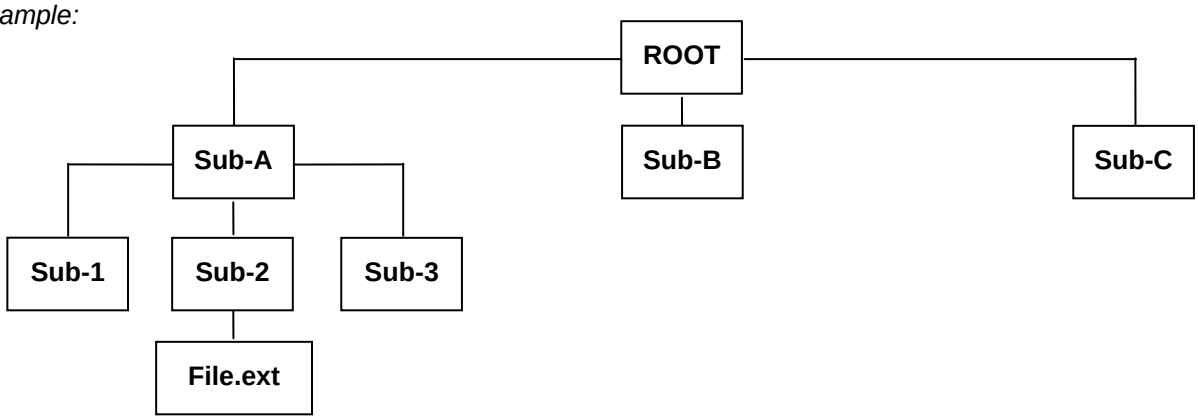
The xsl:template element's *match* attribute tells the processor to look for "stuff_from_source_tree." If it finds it, it should instantiate in the result tree a level-1 XHTML heading whose contents will be "other_stuff" from the source tree. Locating source-tree content is the work of a separate standard — XPath.

Expressions

XPath rules are defined in the form of expressions — strings of characters. For most practical purposes, the term "XPath expression" is a reference to a location, whether in a source file or at a remote location. Expressions are like the atoms of the XPath language; in some cases you can use them in their elemental form, and in many others they will be assembled into compounds.

XPath language defined a syntax for locating or extracting nearly anything within an XML document. The "molecule" of XPath syntax is the location path. A simple location path is a text string — an XPath expression — which resembles a path to a file, either on a local file system or out on the Internet somewhere.

Example:



You can locate a file on the directory tree using a path that looks like this:

```
/ROOT/Sub-A/Sub-2/File.ext
```

This directs an operating system, a Web browser, or some other application to walk down the hierarchy of directories or folders. The first slash set the search at the topmost level of the source tree. Each succeeding slash

Copyright © 2010 by Paul Roberts • All rights reserved

separates a resource (directory or file) at a given level from those above or below it in the hierarchy. A path with this leading slash is called an absolute path. A relative path omits the leading slash, telling the application to start wherever it already is and work down through the directory tree.
Relative paths also may begin with certain special characters: a period (a.k.a. "full stop") or a pair of periods. The single period is generally unnecessary: it simply represents "here" — the directory where you are currently located. The double period is more useful; it means "go to the parent directory."

The structure of a file system is exactly analogous to the structure of an XML document. Both consist of a single room "thing" from which descend a whole family of nodes. Some nodes contain others, and at the very bottom of a given branch of the tree might be a simple empty container or some real content—-a file, in the case of a file system, or text, in the case of an XML document.

Nodes that XPath knows

XPath's location steps are capable of locating much more than just simple boxes (elements) and their contents (other elements and/or text). XPath is able to locate the following kinds of modes in any source document:

root node
elements
text
attributes
processing instructions (PIs)
comments
namespaces

Two things XPath cannot locate are the XML declaration and the document type declaration. This is because the file is already parsed, so XPath does not see those two features as nodes. Instead, it sees them as part of a well formed and valid XML document.

Every node in an XML document has a name. This is plain to see for elements and attributes, which are named in the file. Note that element and attribute names that include a prefix actually consist of three parts: the prefix, the URI associated with the prefix, and the so-called "local name" (the element or attribute name without the prefix).

Consider the following situation:

xlink:href=http://www.moderntimes.com/palace/kc.htm
Prefix = xlink:
URI = http://www.moderntimes.com/palace/kc.htm
local name = href

For PI nodes, the name is the same as the PI target — that is, everything between the opening <? and either the first space, if there is one, or the closing ?>, if not. Namespace nodes constitute a special case. The name of a namespace node is the prefix associated with the namespace, if there is a prefix or an empty string otherwise. The name of a namespace node also includes a URI component, which is always empty.

- *String-values* — A string value of a text node is whatever text it contains. Beyond text, attributes, and comment nodes, there are some other XPath node types and their string values:

- *Root node* — The concatenated string value of all text nodes in the document.

- *Element* — The concatenated string value of all text nodes within the element's start and end tags.
- *PI* — All the text in the processing instruction that follows the PI's target, up to the closing ?>.
- *Namespace* — The URI of the namespace node.

Node Sets

A file system path always locates a single resource — a single directory or file. In contrast, an XPath location finds a node-set — a collection of zero, one, or more nodes.

Example:

Consider the following simple location path:

```
/flixinfo/title
```

Because the root node (/) has only one child node named flixinfo, and the flixinfo element has only one child named "title," this location path locates a single node: the one element whose name is title. More precisely, it locates a node-set that just happens to consist of a single node. On the other hand, the following location path returns a node-set consisting of two nodes — the two releaseyear elements

Position and Context

A member node of a node-set has a position. When you're using relative location paths, it's vitally important that you understand the context in which a given location step is evaluated. This isn't much of an issue when you're using XPath with XPointer. In that case, you'll be accessing a single node-set and not moving around within the node-set. But in XSLT, this notion of context assumes enormous importance because you establish an initial context and them move up, down and perhaps sideways within it.

A simple way of changing context is to think of a simple location path such as this:

```
/flixinfo/cast/leadcast/female
```

Each succeeding location step narrows the context in which all following location steps will be evaluated. As soon as the flixinfo element is names in the location path, the xml-stylesheet PI is excluded from consideration; with cast, all the other children of flixinfo are disregarded; and so on.

As you move around in a node-set, whether in a simple location path or in a more complicated fashion (as in an XSLT stylesheet), there will always be one context node. This is the node in whose context succeeding relative location paths will be evaluated. In the preceding location path, at first flixinfo element is the context node, so that when the processor hits the cast step it already knows which leg of the document three is being walked; then the cast step further narrows the context, becoming the context node in which the leadcast step will be evaluated, and so on.

Full location step and syntax

XPath specifies a rich set of options for building any location step. These options are summed up in the full syntax for a location step, which looks something like this:

```
axis::nodeTest[predicate]
```

The axis and predicate portions of this syntax are optional. If you omit the axis, you also omit the double colon; and if you omit the predicate, you also omit the square brackets surrounding it. The general purpose of each of these components is as follows:

1. The axis tells the XPath-aware application "which direction to look" from the context node of the moment. Each axis specifies some relationship to the context node, and there's a default: to consider only children of the context node.

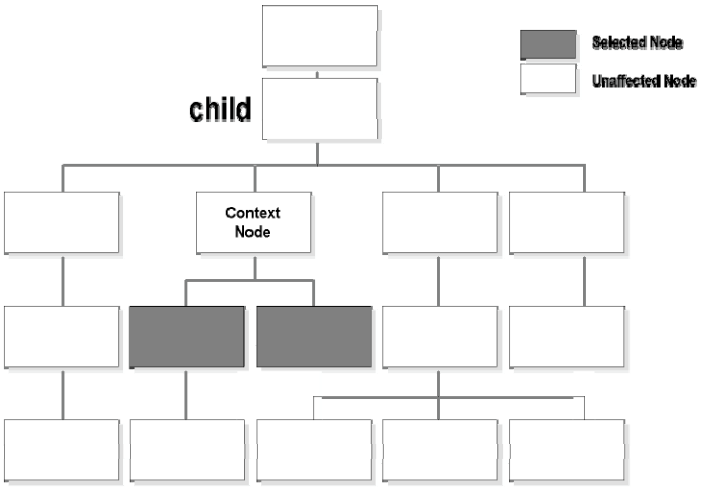
   The optional axis portion of a location step directs the application to look up, down or sideways from the context mode. These directions are based on a view of all the nodes in a document as something like a family.

   Remember that when you use an axis, you separate its name from the node test with double colons (::). There are 13 axes that may be used in a location step:

child	self	following-sibling	preceding
parent	descendant-or-self	preceding-sibling	attribute
descendant	ancestor-or-self	following	namespace
ancestor			

   Each axis is classified as either a forward axis or a reverse axis, depending on which way you have to look along that particular axis. The significance of this is that if an axis is a forward axis, the processor will see matching nodes in document order relative to the context mode (i.e., the order in which they appear physically in the document); if the axis is a reverse axis, the processor sees any matching modes in reverse document order relative to the context mode.

   The ancestor, ancestor-or-self, preceding-sibling, and preceding axes are reverse axes. All others are considered forward axes.



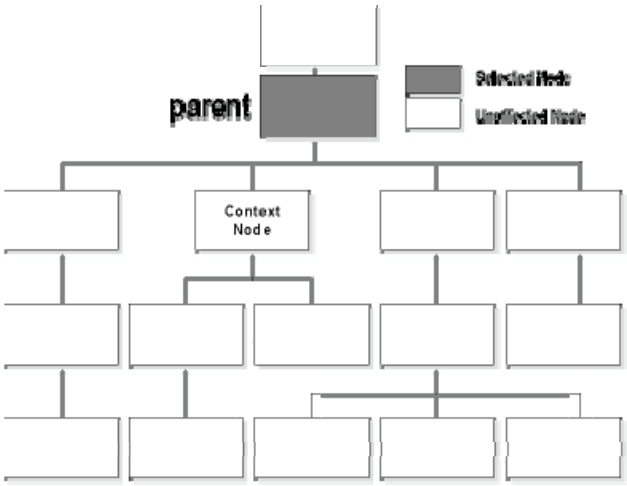
   child axis — This is the default axis. It points the processor down in the hierarchy, one level. The context node must be the root node or an element in order to look along the child axis. (That is, if the context node is an attribute or PI, looking along the child axis will always return an empty node-set.)

The child axis is restricted in that it enables the processor to see only elements, PIs, comments and text nodes. Attributes and namespace nodes are invisible it you're looking at the context node's children. For obvious reasons, the root node is never located on the child axis, not matter what the context node is. If the root node is the context node, it always has a least one child, which is the document's root element.

Example — Here is a sample XSLT template rule that uses the child axis in a couple of places. (Remember that because this is the default axis, you can omit the child:: from either of these XPath expressions.)

```
<xsl:template match="/child::flixinfo"> <!-- The slash sets the context mode as absolute -->
<html>
<head>
<title>
FlixML Review: <xsl:value-of select="child::title"/>
<-- The value-of rule defines a relative mode. The context node changed within the template rule. It will remain the
same throughout the template unless another rule is set within the template. At the time the xsl:value-of element's
select attribute is evaluated, the context node is the root flixinfo element. That's why we can use a relative location
step here, rather than having to repeat the entire context that's already been established.  -->
</title>
</head>
  </html>
</xsl:template>
```

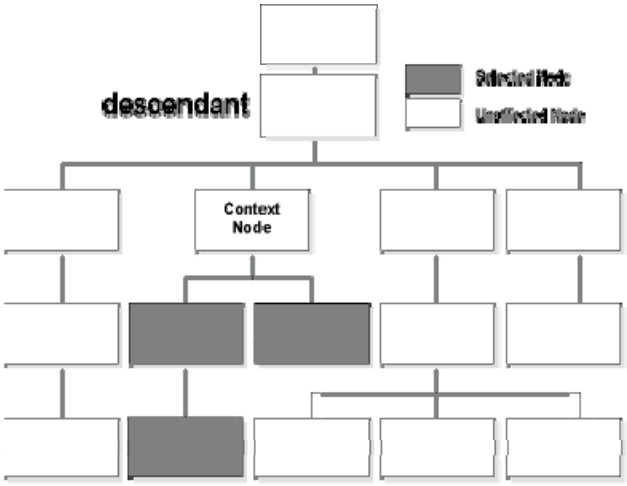
This template rule begins by selecting the root node's child whose name is "flixinfo." When it finds a matching portion of the source tree, it instantiates in the result tree some XHTML elements: the root `html`, a `head`, and a `title` element within the head. The value that's inserted in the result tree's title element consists of some of some literal text ("FlixML Review: ") and the value of the context node's child named "title." Thus, a browser viewing the result tree from this transformation will display (in this case) the string "FlixML Review: Criss Cross" in the title bar of its window.



parent axis — This is the inverse of the child axis. It locates the element (or root node) immediately above the context node in the document's node tree. The root node obviously has no parent, but every other node in the document has exactly one parent. Although attribute and namespace nodes can never serve

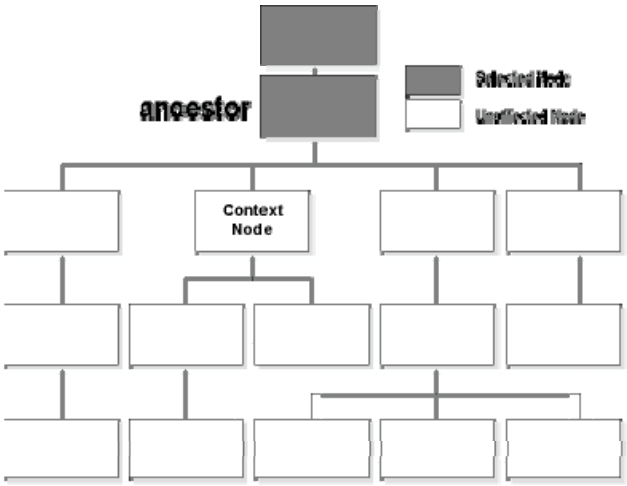
as children, they do have parents: the elements which declare their values. No nodes other than elements or the root node can be located along the parent axis.

The `parent::` designation for an axis is functionally the same as the double dots (..) notation in a location step.

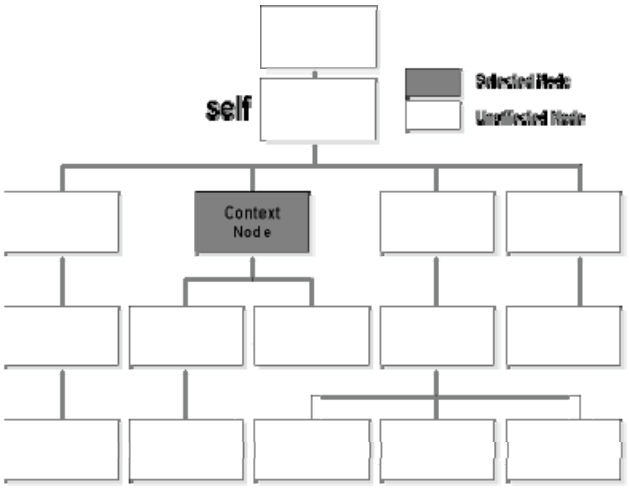


descendent axis — You can locate nodes further down than one level with the descendent axis. This extends the child axis indefinitely to the lowest level of that particular branch of the document tree. All the restrictions that applied to the child axis are also in place here. That is, the root node is never visible along the descendent axis and neither attributes nor namespace nodes can be located with it.

When using the descendent axis, it's sometimes desirable to locate an entire sub-tree — all descendents of a particular node.



ancestor axis — This works as the inverse of the descendent axis by extending the parent up through the document tree. As with the parent axis, the ancestor locates elements and the root node only. The root node will be located on the ancestor axis from any other node in the document, and the root node itself has no ancestors.

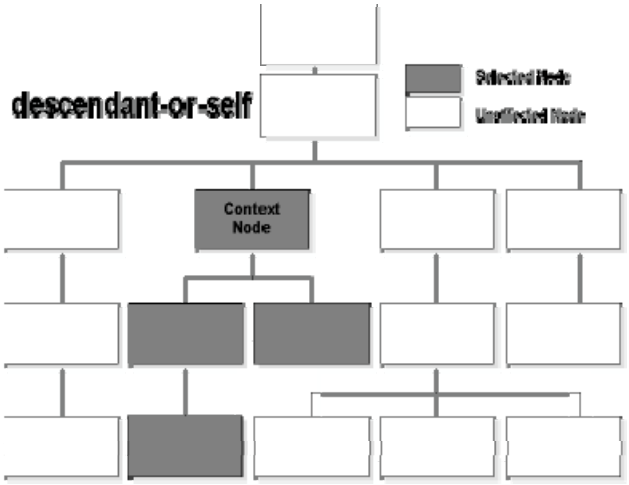


self axis — It may not be immediately obvious why you'd ever need something like the self axis, along which you can see only one node — the context node itself. After all, you're already there.

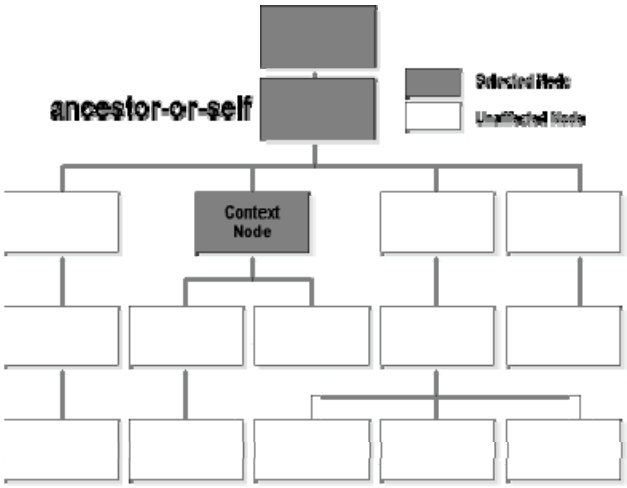
The most likely place you'll find yourself using the self axis is within a template rule, in the value of an *xsl:value-of* element's select attribute. For instance, you could do something like this:

```
<xsl:template match="/descendent::comment()">
Here's a comment for you: <xsl:value-of select="self::comment()"/>
</xsl:template>
```

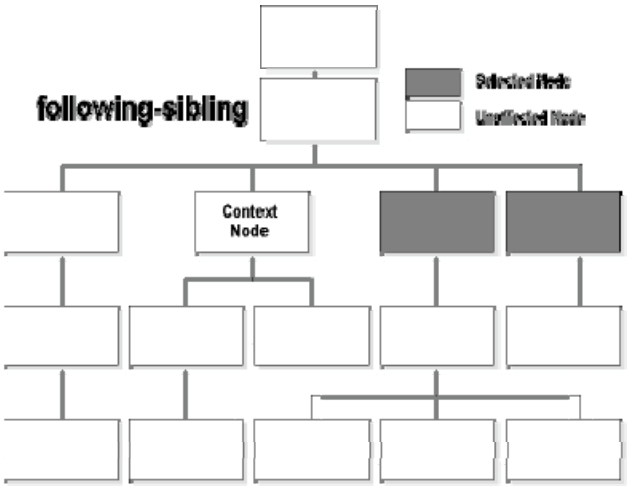
This would instantiate in the result tree the text string, "Here's a comment for you: Note: cross-check studio name".



descendent-or-self axis — This axis extends the notion of the descendent axis by adding one more node to the resulting node-set: the context node itself. This is important because it ensures that you're getting an entire sub-tree. The axis is so useful that there's a shortcut for it: a pair of adjacent slashes (//).

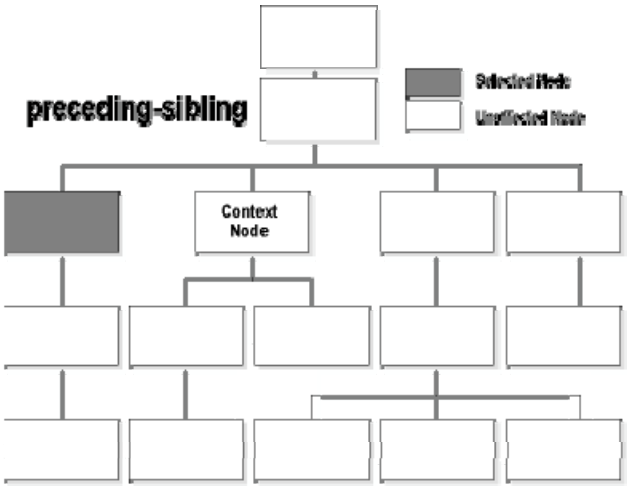


ancestor-or-self axis — It locates all nodes along the ancestor from the context node, plus the context node itself. No shortcuts here.



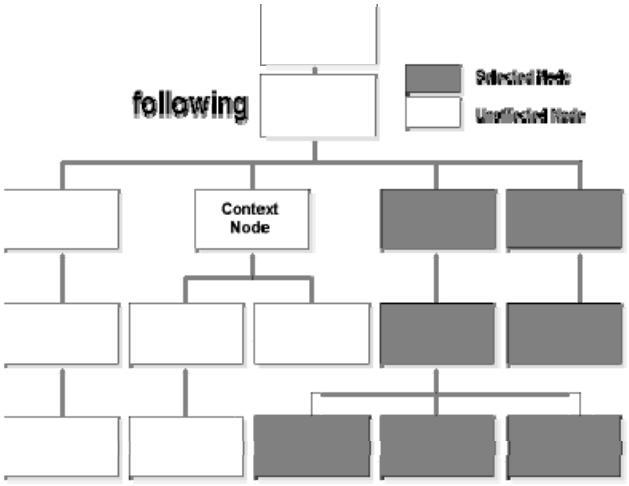
following-sibling axis — This locates all nodes that share the same parent as the context node and that come after the context node, in document order. If the context node is the cinamatog element in our sample document, you can locate the editor, score and costumer elements at once with a location path such as this:

```
following-sibling::*
```

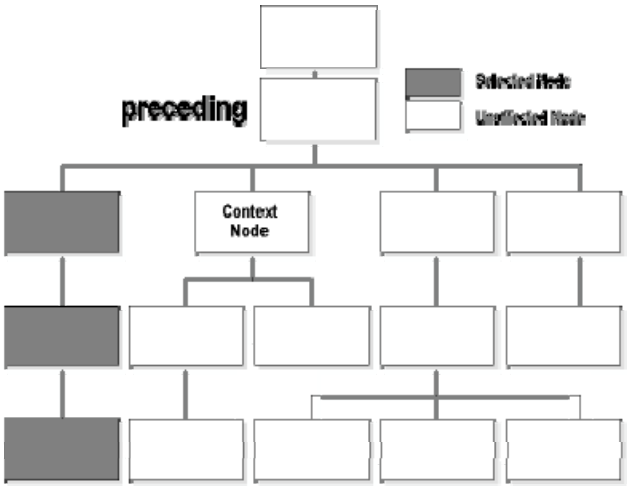


preceding-sibling axis — This locates children of the same parent as the context node itself — but only those which some before the context node, in document order. Again, assuming the context node is the cinamatog element, this path locates the director and two screenwriter elements:

```
preceding-sibling::*
```



following axis — Locates all nodes in the document — regardless of parent — which follow the context node, in document order. Although is includes all nodes as in the following-sibling axis, it explicitly excludes attribute and namespace nodes. In order for a node to be visible along the following axis, it must both start and end after the end of the context node itself. There can't be any overlap.



preceding axis — Finds all nodes in the document (except attribute and namespace nodes) that bother start and end before the start of the context node itself, including any nodes you could find along the preceding-sibling axis.

attribute axis — This is one of two XPath axes that have no real counterpart in a family tree or bloodline. If the context node is an element, looking along the attribute axis will locate that element's attributes; if the context node is not an element, looking along the attribute axis will locate an empty node-set. You can use a specific attribute name to refer to a specific attribute of the context node, or an asterisk wildcard to refer to all its attributes.

namespace axis — Locates all namespace nodes visible from a given context node. If the context node is anything other than an element, the resulting node-set is empty.

2. The node test directs the XPath-aware application's attention to a particular node, or class of nodes, along the indicated axis.

   You have a few choices here. You can name the desired node. You can locate a node by its type. Here are the types:

processing-instruction()	Locates a PI node, regardless of its name.
processing-instruction(name)	Locates a PI node whose name is name.
comment()	Locates a comment node.
text()	Locates a text node.
node()	Locates a node of any type.

   a) If the context node is the root node, either of the following location steps will find the xml-stylesheet PI:

      ```
      processing-instruction()
      processing-instruction("xml-stylesheet")
      ```

   b) Assume that the context node is the remarks element. You can locate the document's lone comment with this location path:

```
../comment()
```

The leading dots (..) take you up the hierarchy to the remarks element's parent, or the flixinfo element; the next location step in the path leads you down to that element's child or children of the comment type.

c) Note that the content model for the plotsummary element is a mixed content model; it contains some text of its own and (optionally) some empty parabreak elements and/or emph elements, the latter with text-node children of their own. If the context node is the flixinfo element, and the location path is:

```
plotsummary/text90
```

then you locate all of the text nodes which are children of the *plotsummary* element, but none of the *parabreak* elements, and none of the *emph* elements (or their text nodes).

d) If the context node is the genre element, the following location path locates a node-set whose members are the xml-stylesheet PI and the flixinfo element:

```
../node()
```

Wildcards in the node test — In addition to matching on a specific name or a node type, you can also use an asterisk (* and pronounced as'-ter-iSK not as'-ter-riks) as the node test. This is a special form of the name node test; it says to locate a node (element or attribute only, depending on the axis) regardless of its name.

If the context node is one of the releaseyear elements in the sample document, you might use a relative location path such as this:

```
../genre/*
```

This locates the primarygenre and othergenre elements.

3. The *predicate* narrows the selection even further, by indicating some condition a node must meet in order for it to be considered a "match." It is tacked on the end of a location step, as a value surrounded by square brackets and following the node test.

What goes inside the square brackets is a logical or Boolean test. The predicate's test is applied to every node that matches the attribute/node test condition. If the result of the predicate's test is true for a given node, then the node is selected; if false, then the node is disregarded.

Here is how we could code a couple of template rules for handling the Criss Cross review's two *releaseyear* elements, using predicates as part of their respective paths:

```
<xsl:template match="//releaseyear[@role='initial']">
<-- @ shortcut for attribute axis -->
<h3><xsl:value-of select="."/></h3>
</xsl:template>

<xsl:template match="//releaseyear[@role='alt']">
```

```
<h4><xsl:value-of select="."/></h4>
</xsl:template>
```

The *match* attribute in the first template rule says, “Instantiate the following content in the result tree only when you find a *releaseyear* element in the document with a role attribute whose value is initial.” When such a *releaseyear* element is located, its content is displayed as an XHTML level-3 heading. The second instance displays the contents in a level-4 heading.

Here are the logical operators that can be used in a location step’s predicate:

Operator	Meaning
=	Equals
!=	Does not equal
<	Is less than
>	Is greater than
<=	Is less than or equal to
>=	Is greater than or equal to

Predicate: Special Cases — XPath considers all possible nodes in a document tree to have a value of true or false — true if the node actually exists, false otherwise. Therefore, within a predicate you can simply include an XPath location path/step, which is interpreted as saying, “If the node identified in this predicate exists.”

Example — If we wanted our stylesheet to show only bad reviews of the film, we could construct a template rule like this:

```
<xsl:template match="//otherreview[badreview]">
<!-- Moving along the descendent-or-self axis, we match each otherreview  element
 for which a badreview child exists. -->
```

Another special case involves a predicate may have (or evaluate to) a numeric value. The processor treats this as meaning, “If the node is at the specified position in the node-set returned by this location step.” For instance, you can locate the first *castmember* element in the document with a location path such as this:

```
//castmember[1]
<!-- The above is shorthand for the following:-->
//castmember[position()=1]
```

You don’t have to use a literal there. You could include some calculated value, or a value returned by one of the XPath functions.

Compound predicates — You can test for more than one condition in the predicate by chaining them together, separated by the keywords `and` and `or`. Thus, the following selects a *plotsummary* element only if it has both a *parabreak* and an *emph* child:

```
//plotsummary[parabreak and emph]
```

On the other hand, the following selects a *plotsummary* element if it has a *parabreak* or an *emph* child:

```
//plotsummary[parabreak or emph]
```

Compound predicates can get quite elaborate when you're testing for more than two conditions and using both `and` and `or`. In these cases, it's generally helpful to use parentheses to group together the conditions you want to be tested together. For instance, consider the following code:

```
//castmember[parent::female or (ancestor::leadcast and parent::male)]
```

This locates a node-set consisting of all *castmember* elements for which either of the following conditions is true: 1) The parent of the *castmember* element is a female element; or 2) the *castmember* element has a *leadcast* ancestor and a male parent.

Now change the grouping as follows:

```
//castmember[(parent::female or ancestor::leadcast) and parent::male)]
```

This locates a node-set consisting of all *castmember* elements for which the following conditions are true: 1) The parent of the *castmember* element is a female element or the *castmember* has a *leadcast* ancestor; and 2) the parent of the *castmember* element is a male element.

It is easier to begin working with the simplest condition. Then you can start building a more complex condition to get at the information you want to define in the template rule.

You can select node-sets from two or more completely different location paths, allowing you to assemble a view of your source document's content that bears little if any relationship to its actual structure. You accomplish this by chaining together the separate location paths using the so-called pipe or vertical bar character (|). For example, if you want the same template rule to apply to the *plotsummary*, *goodreview*, *badreview* and *remarks* elements in the Criss Cross review, you could do something like this:

```
<xsl:template
match="//plotsummary | //goodreveiw | //badreview | //remarks">
<p><xsl:value-of select="."/></p>
</xsl:template>
```

XPath Functions

XPath isn't restricted to navigating up and down and sideways in a document tree. It enables you to locate content in other than its pure form and to obtain certain kinds of information about the content other than simple where it lies within the document's structure.

The XPath spec establishes several categories of functions that may be used in XPath location steps in various ways. Each category is names according to the kind of value returned by or passed to the functions in the category. Here are the fours categories:

- node-set
- string
- Boolean
- numeric

You can also perform a number of straight arithmetic operations that return a single value in the same way that functions do.

Node-Set Functions

Function call	Returns	Description
`last()`	number	Returns the number of nodes in the current node-set.
`position()`	number	Returns the position of the context node within the current node-set.
`count(`*`nodeset`*`)`	number	Returns the number of nodes in the *nodeset* argument.
`is(`*`value`*`)`	node-set	Returns the element node whose ID-type attribute has the value specified in the *value* argument.
`local-name(`*`nodeset?`*`)`	string	If *nodeset* argument is passed, returns a string representing the local name (that is, without a namespace prefix) of the first node in that node-set; if *nodeset* is omitted, returns the local name of the context node.
`namespace-uri(`*`nodeset?`*`)`	string	If *nodeset* argument is passed, returns a string representing the namespace URI associated with the name of the first node in that node-set; if *nodeset* is omitted returns the namespace URI associated with the context node's name. Note that this may be an empty string, if the name isn't associated with any particular namespace URI.
`name(`*`nodeset?`*`)`	string	If *nodeset* argument is passed, returns a string representing the entire name (that is, including any namespace prefix) of the first node in that node-set; if *nodeset* is omitted, returns the name of the context node (including namespace prefix, if any).

All the functions in this category either operate on or return an XPath node-set. You will probably use *last()*, *position()*, and *count()* most often. They're all closely related because they deal with the number of nodes in a node-set. Note: There is no *first()* function because you can always find the first node in a node-set by using a numeric predicate with a value of 1. For example:

```
//distributor[1]
```

The *position()* function can be compared to some value in a predicate simply by using the value alone. Thus, these two predicates return the same result:

```
[position()=4]
[4]
```

Both *last()* and *count()* functions return the size of the node-set. However, the *count()* function takes and argument while the other one does not.

Here are a couple of examples of how you could use these three functions:

```
Displaying item <xsl:value-of select="position()"/> of
<xsl:value-of select="last()"/> items found.
There are <xsl:value-of select="count(//goodreview)"/> good reviews and
<xsl:value-of select="count(//badreview)"/> bad reviews of this title.
```

String Functions

Function call	Returns	Description
`string(value?)`	string	Converts *value* I argument to a string and returns the result. If *value* is a node-set, it returns the string-value of the first node in the node-set. If *value* is omitted, returns the string-value of the context node.
`concat(str1, str2…)`	string	Combines multiple strings into one big string, and returns the result.
`starts-with(str1, str2)`	Boolean	If the value of *str1* begins with the value of *str2*, returns the value true; otherwise returns the value false.
`contains(str1, str2)`	Boolean	If the string is passed as *str1* contains the string passed as *str2*, returns true; otherwise returns false.
`substring(str, nstart, nchars)`	string	Returns a portion of the *str* argument, starting with the character at position *nstart*; if *nchars* argument is passed, the result will contain that many characters, otherwise it will include all characters in *str* from *nstart* to the end.
`substring-before(str1, str2)`	string	Returns the portion of *str1* which appears before the string passed as *str2*.
`substring-after(str1, str2)`	string	Returns the portion of *str1* which appears after the string passed as *str2*.
`string-length(string?)`	number	Returns the number of characters in *string*; if *string* is omitted, returns the number of characters in the string-value of the context node.
`normalize-space(string?)`	string	Strips leading and trailing white space from *string*, and converts multiple adjacent white space characters within *string* to a single space; and returns the result; it *string* is omitted, returns the result of this operation on the string-value of the context node.
`translate(str1, str2, str3)`	string	Replaces individual characters appearing in both *str1* and *str2* with other characters from *str3*.

Boolean Functions

Function call	Description
`boolean(value)`	Converts *value* to true or false.
`not(Boolean)`	If *Boolean* is true, returns false, and vice versa.
`true()`	Returns the value true.
`false()`	Returns the value false.
`lang(string)`	If the language in which the context node is presented (as determined by any xml:lang attributes in scope for the context node) is the same as the language represented by *string*, returns true; otherwise returns false.

Numeric Functions

Function call	Description
`number(value?)`	Converts *value* to numeric form, for use in calculations, numeric sorting, and so on. Boolean true converts to the value 1; false, to 0. If *value* is a node-set, it is first converted to its string-value to a number. If *value* cannot be converted to a number, returns the special value *NaN* (for “not a number”).
`sum(nodeset)`	Returns the sum of all nodes in the passed nodeset argument, after converting each to a number.
`floor(number)`	Returns the largest integer which is less than or equal to number.
`ceiling(number)`	Returns the smallest integer which is greater than or equal to number.
`round(number)`	Returns the integer whose value is closest to number. If there are two such numbers (i.e., number contains a decimal portion of .5), returns the higher of the two numbers.

Numeric/Arithmetic Operators

Operator	Description
num1 `+` *num2*	Adds *num1* to *num2*, returning the sum.
num1 `-` *num2*	Subtracts *num1* from *num2*, returning the difference.
num1 `*` *num2*	Multiplies *num1* and *num2*, returning the product.
num1 `div` *num2*	Divides *num1* by *num2*, returning the quotient.
num1 `mod` *num2*	Returns the remainder of dividing *num1* by *num2*.

General Guidelines

#1: Avoid the leading // in XSLT style sheets

The reason from this guideline is hard to explain, but it has to do with processing efficiency. In an XSLT style sheet, most template rules establish their own within a context established by other template rules. Using a leading double dash is horribly inefficient.

#2: Simple is usually better…

There are likely to be a thousand routes to any given data item, but the shortest, most direct route is probably to be the most efficient and practically workable one. You might be called upon to explain what your code does. The simpler the code is, the easier the explanation will be.

#3: …But don't fear the complex

XPath's role is to make it impossible not to find something in an XML document. Most often you will be able to find data easily, but occasionally you may need to perform some tricks to get the desired results. If something makes you nervous, play with it until you get it.

#4: Locate by structure, not by content

If you catch yourself looking for a particular string in an XML document, you often will come up empty handed. Instead, look for what is there and let the structure determine the content you find.