

- | | |
|--------------------------------|-----------------------|
| 1. XPath expression | 7. Function Types are |
| 2. XPath Model | A. Node-set |
| 3. Template and Context | B. Positional |
| 4. Basic Addressing | C. Numeric |
| a. indexing | D. Boolean |
| b. wild card | E. String-Valued |
| c. function | F. NameSpace |
| d. extended XPATH addressing | G. Conversion |
| 5. XPATH datatype and Operator | |
| 6. String value element | |

How XPath Works

The XPath specification is the foundation for a variety of specifications, including XSLT and linking/addressing specifications such as XPointer. So an understanding of XPath is fundamental to a lot of advanced XML usage. This section provides a thorough introduction to XPath in the context of XSLT so that you can refer to it as needed.

Note: In this tutorial, you won't actually use XPath until later, in the section, Transforming XML Data with XSLT (page 287). So, if you like, you can skip this section and go on ahead to the next section, Writing Out a DOM as an XML File (page 265). (When you get to the end of that section, there will be a note that refers you back here so that you don't forget!)

XPath Expressions

In general, an XPath expression specifies a *pattern* that *selects* a set of *XML nodes*. XSLT templates then use those patterns when applying transformations. (XPointer, on the other hand, *adds* mechanisms for defining a *point* or a *range* so that XPath expressions can be used for addressing.)

The nodes in an XPath expression refer to more than just elements. They also refer to text and attributes, among other things. In fact, the XPath specification defines an abstract document model that defines seven kinds of nodes:

- 1 • Root ✓
- 2 • Element ✓
- 3 • Text ✓
- 4 • Attribute ✓
- 5 • Comment ✓
- 6 • Processing instruction ✓
- 7 • Namespace ✓

Note: The root element of the XML data is modeled by an *element* node. The XPath root node contains the document's root element as well as other information relating to the document.

OK

The XSLT/XPath Data Model

Like the Document Object Model, the XSLT/XPath data model consists of a tree containing a variety of nodes. Under any given element node, there are text nodes, attribute nodes, element nodes, comment nodes, and processing instruction nodes.

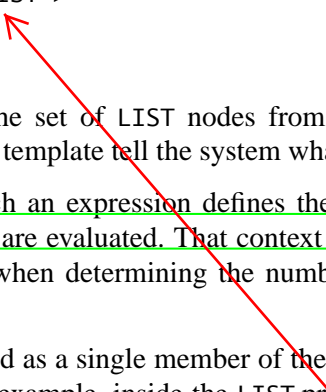
In this abstract model, syntactic distinctions disappear, and you are left with a normalized view of the data. In a text node, for example, it makes no difference whether the text was defined in a CDATA section or whether it included entity references. The text node will consist of normalized data, as it exists after all parsing is complete. So the text will contain a < character, whether or not an entity reference such as `<`; or a CDATA section was used to include it. (Similarly, the text will contain an `&` character, whether it was delivered using `&`; or it was in a CDATA section.)

In this section, we'll deal mostly with element nodes and text nodes. For the other addressing mechanisms, see the XPath specification.

Templates and Contexts

An XSLT *template* is a set of formatting instructions that apply to the nodes selected by an XPath expression. In a stylesheet, an XSLT template would look something like this:

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



The expression `//LIST` selects the set of `LIST` nodes from the input stream. Additional instructions within the template tell the system what to do with them.

The set of nodes selected by such an expression defines the context in which other expressions in the template are evaluated. That context can be considered as the whole set—for example, when determining the number of the nodes it contains.

The context can also be considered as a single member of the set, as each member is processed one by one. For example, inside the LIST-processing template, the expression `@type` refers to the `type` attribute of the current `LIST` node. (Similarly, the expression `@*` refers to all the attributes for the current `LIST` element.)

Basic XPath Addressing

An XML document is a tree-structured (hierarchical) collection of nodes. As with a hierarchical directory structure, it is useful to specify a *path* that points to a particular node in the hierarchy (hence the name of the specification: XPath). In fact, much of the notation of directory paths is carried over intact:

- The forward slash (`/`) is used as a path separator.
- An absolute path from the root of the document starts with a `/`.
- A relative path from a given location starts with anything else.
- A double period (`..`) indicates the parent of the current node.
- A single period (`.`) indicates the current node.

For example, In an Extensible HTML (XHTML) document (an XML document that looks like HTML but is *well formed* according to XML rules), the path `/h1/h2/` would indicate an `h2` element under an `h1`. (Recall that in XML, **element names are case-sensitive**, so this kind of specification works much better in XHTML than it would in plain HTML, **because HTML is case-insensitive**.)

In a pattern-matching specification such as XPath, the specification `/h1/h2` selects **all h2 elements** that lie under an h1 element. To select a specific **h2 element**, you use square brackets `[]` for indexing (like those used for arrays). The path `/h1[4]/h2[5]` would therefore select the fifth h2 element under the fourth h1 element.

Note: In XHTML, all element names are in lowercase. That is a fairly common convention for XML documents. However, uppercase names are easier to read in a tutorial like this one. So for the remainder of the XSLT tutorial, all XML element names will be in uppercase. (Attribute names, on the other hand, will remain in lowercase.)

A name specified in an XPath expression **refers to an element**. For example, h1 in `/h1/h2` refers to an h1 element. To refer to an attribute, you prefix the attribute name with an `@` sign. For example, `@type` refers to the type attribute of an element. Assuming that you have an XML document with LIST elements, for example, the expression `LIST/@type` selects the type attribute of the LIST element.

Note: Because the expression does not begin with `/`, the reference specifies a `list` node relative to the current context—whatever position in the document that happens to be.

Basic XPath Expressions

The full range of XPath expressions takes advantage of the wildcards, operators, and functions that XPath defines. You'll learn more about those shortly. Here, we look at a couple of the most common XPath expressions simply to introduce them.

The expression `@type="unordered"` specifies an attribute named type whose **value** is unordered. As you know, an expression such as `LIST/@type` specifies the type attribute of a LIST element.

You can combine those two notations to get something interesting! In XPath, the square-bracket notation (`[]`) normally associated with indexing is extended to specify selection criteria. So the expression `LIST[@type="unordered"]` selects all LIST elements whose type value is unordered.

Similar expressions exist for elements. Each element has an associated *string-value*, which is formed by concatenating all the text segments that lie under the

element. (A more detailed explanation of how that process works is coming up in String-Value of an Element, page 261.)

Suppose you model what's going on in your organization using an XML structure that consists of PROJECT elements and ACTIVITY elements that have a text string with the project name, multiple PERSON elements to list the people involved and, optionally, a STATUS element that records the project status. Here are other examples that use the extended square-bracket notation:

- /PROJECT[.="MyProject"]: Selects a PROJECT named "MyProject"
- /PROJECT[STATUS]: Selects all projects that have a STATUS child element
- /PROJECT[STATUS="Critical"]: Selects all projects that have a STATUS child element with the string-value Critical

Combining Index Addresses

The XPath specification defines quite a few addressing mechanisms, and they can be combined in many different ways. As a result, XPath delivers a lot of expressive power for a relatively simple specification. This section illustrates other interesting combinations:

- LIST[@type="ordered"][3]: Selects all LIST elements of type ordered, and returns the third
- LIST[3][@type="ordered"]: Selects the third LIST element, but only if it is of type ordered

Note: Many more combinations of address operators are listed in section 2.5 of the XPath specification. This is arguably the most useful section of the spec for defining an XSLT transform.

Wildcards

By definition, an unqualified XPath expression selects a set of XML nodes that matches that specified pattern. For example, /HEAD matches all top-level HEAD entries, whereas /HEAD[1] matches only the first. Table 7-1 lists the wildcards

that can be used in XPath expressions to broaden the scope of the pattern matching.

Table 7–1 XPath Wildcards

Wildcard	Meaning
*	Matches any <u>element node</u> (<u>not attributes or text</u>)
node()	Matches <u>any node</u> of any kind: <u>element node, text node, attribute node, processing instruction node, namespace node, or comment node</u>
@*	Matches <u>any attribute node</u>

In the project database example, `/*/PERSON[.="Fred"]` matches any PROJECT or ACTIVITY element that names Fred. ✓

Extended-Path Addressing

So far, all the patterns you’ve seen have specified an exact number of levels in the hierarchy. For example, `/HEAD` specifies any HEAD element at the first level in the hierarchy, whereas `/*/*` specifies any element at the second level in the hierarchy. To specify an indeterminate level in the hierarchy, use a double forward slash (//). For example, the XPath expression `//PARA` selects all paragraph elements in a document, wherever they may be found.

The `//` pattern can also be used within a path. So the expression `/HEAD/LIST//PARA` indicates all paragraph elements in a subtree that begins from /HEAD/LIST.

XPath Data Types and Operators

XPath expressions yield either a set of nodes, a string, a Boolean (a true/false value), or a number. Table 7–2 lists the operators that can be used in an XPath expression

Table 7–2 XPath Operators

Operator	Meaning
	Alternative. For example, <code>PARA LIST</code> selects all <code>PARA</code> and <code>LIST</code> elements.
or, and	Returns the or/and of two Boolean values.
=, !=	Equal or not equal, for Booleans, strings, and numbers.
<, >, <=, >=	Less than, greater than, less than or equal to, greater than or equal to, for numbers.
+, -, *, div, mod	Add, subtract, multiply, floating-point divide, and modulus (remainder) operations (e.g., <code>6 mod 4 = 2</code>)

Expressions can be grouped in parentheses, so you don't have to worry about operator precedence.

Note: *Operator precedence* is a term that answers the question, “If you specify `a + b * c`, does that mean `(a+b) * c` or `a + (b*c)`?” (The operator precedence is roughly the same as that shown in the table.)

String-Value of an Element

The string-value of an element is the concatenation of all descendent text nodes, no matter how deep. Consider this mixed-content XML data:

```
<PARA>This paragraph contains a <B>bold</B> word</PARA>
```

The string-value of the `<PARA>` element is `This paragraph contains a bold word`. In particular, note that `` is a child of `<PARA>` and that the text `bold` is a

child of . The point is that all the text in all children of a node joins in the concatenation to form the string-value.

Also, it is worth understanding that the text in the abstract data model defined by XPath is fully normalized. So whether the XML structure contains the entity reference < or < in a CDATA section, the element's string-value will contain the < character. Therefore, when generating HTML or XML with an XSLT stylesheet, you must convert occurrences of < to < or enclose them in a CDATA section. Similarly, occurrences of & must be converted to &.

XPath Functions

This section ends with an overview of the XPath functions. You can use XPath functions to select a collection of nodes in the same way that you would use an element specification such as those you have already seen. Other functions return a string, a number, or a Boolean value. For example, the expression `/PROJECT/text()` gets the string-value of `PROJECT` nodes.

Many functions depend on the current context. In the preceding example, the *context* for each invocation of the `text()` function is the `PROJECT` node that is currently selected.

There are many XPath functions—too many to describe in detail here. This section provides a brief listing that shows the available XPath functions, along with a summary of what they do.

Note: Skim the list of functions to get an idea of what's there. For more information, see section 4 of the XPath specification.

Node-Set Functions

Many XPath expressions select a set of nodes. In essence, they return a *node-set*. One function does that, too.

- `id(...)`: Returns the node with the specified ID.

(Elements have an ID only when the document has a DTD, which specifies which attribute has the ID type.)

So, i dont want this one..since schema only my choice

Positional Functions

These functions return positionally based numeric values.

- `last()`: Returns the index of the last element. For example, `/HEAD[last()]` selects the last HEAD element.
- `position()`: Returns the index position. For example, `/HEAD[position() <= 5]` selects the first five HEAD elements.
- `count(...)`: Returns the count of elements. For example, `/HEAD[count(HEAD)=0]` selects all HEAD elements that have no subheads.

String Functions

These functions operate on or return strings.

- `concat(string, string, ...)`: Concatenates the string values.
- `starts-with(string1, string2)`: Returns true if *string1* starts with *string2*.
- `contains(string1, string2)`: Returns true if *string1* contains *string2*.
- `substring-before(string1, string2)`: Returns the start of *string1* before *string2* occurs in it.
- `substring-after(string1, string2)`: Returns the remainder of *string1* after *string2* occurs in it.
- `substring(string, idx)`: Returns the substring from the index position to the end, where the index of the first char = 1.
- `substring(string, idx, len)`: Returns the substring of the specified length from the index position.
- `string-length()`: Returns the size of the context node's string-value; the *context node* is the currently selected node—the node that was selected by an XPath expression in which a function such as `string-length()` is applied.
- `string-length(string)`: Returns the size of the specified string.
- `normalize-space()`: Returns the normalized string-value of the current node (no leading or trailing whitespace, and sequences of whitespace characters converted to a single space).
- `normalize-space(string)`: Returns the normalized string-value of the specified string.

- `translate(string1, string2, string3)`: Converts *string1*, replacing occurrences of characters in *string2* with the corresponding character from *string3*.

Note: XPath defines three ways to get the text of an element: `text()`, `string(object)`, and the string-value implied by an element name in an expression like this: `/PROJECT[PERSON="Fred"]`.

Boolean Functions

These functions operate on or return Boolean values.

- `not(...)`: Negates the specified Boolean value.
- `true()`: Returns true.
- `false()`: Returns false.
- `lang(string)`: Returns true if the language of the context node (specified by `xml:Lang` attributes) is the same as (or a sublanguage of) the specified language; for example, `Lang("en")` is true for `<PARA_xml:Lang="en">...</PARA>`.

Numeric Functions

These functions operate on or return numeric values.

- `sum(...)`: Returns the sum of the numeric value of each node in the specified node-set.
- `floor(N)`: Returns the largest integer that is not greater than *N*.
- `ceiling(N)`: Returns the smallest integer that is not less than *N*.
- `round(N)`: Returns the integer that is closest to *N*.

Conversion Functions

These functions convert one data type to another.

- `string(...)`: Returns the string value of a number, Boolean, or node-set.
- `boolean(...)`: Returns a Boolean value for a number, string, or node-set (a non-zero number, a nonempty node-set, and a nonempty string are all true).

- `number(...)`: Returns the numeric value of a Boolean, string, or node-set (true is 1, false is 0, a string containing a number becomes that number, the string-value of a node-set is converted to a number).

Namespace Functions

These functions let you determine the namespace characteristics of a node.

- `local-name()`: Returns the name of the current node, minus the namespace prefix.
- `local-name(...)`: Returns the name of the first node in the specified node set, minus the namespace prefix.
- `namespace-uri()`: Returns the namespace URI from the current node.
- `namespace-uri(...)`: Returns the namespace URI from the first node in the specified node-set.
- `name()`: Returns the expanded name (URI plus local name) of the current node.
- `name(...)`: Returns the expanded name (URI plus local name) of the first node in the specified node-set.

Summary

XPath operators, functions, wildcards, and node-addressing mechanisms can be combined in wide variety of ways. The introduction you've had so far should give you a good head start at specifying the pattern you need for any particular purpose.

Writing Out a DOM as an XML File

After you have constructed a DOM—either by parsing an XML file or building it programmatically—you frequently want to save it as XML. This section shows you how to do that using the Xalan transform package.

Using that package, you'll create a transformer object to wire a DOMSource to a StreamResult. You'll then invoke the transformer's `transform()` method to write out the DOM as XML data.