XSLT Reference

John W. Shipman

2013-07-30 11:50

Abstract

Describes the XSLT language for transforming XML documents.

This publication is available in Web form[1] and also as a PDF document[2]. Please forward any comments to **tcc-doc@nmt.edu**.

Table of Contents

[1] http://www.nmt.edu/tcc/help/pubs/xslt/
[2] http://www.nmt.edu/tcc/help/pubs/xslt/xslt.pdf

1. What is XSLT?

XSLT is a tool for transforming an XML (eXtended Markup Language) document into either an HTML document, or into an XML document of a different document type.

This document assumes that you are familiar with the structure of XML documents; if you are unfamiliar with XML, see the XML help page[3].

Many of the examples use HTML; for reference, see *Building web pages with XHTML 1.1*[4].

Online files related to this document:

- `model.xsl`[5]: This file is the skeleton of an XSLT stylesheet for converting an XML document to HTML. To start a new stylesheet, make a copy of this file and add your title and templates.
- `xslt.xml`[6]: The XML DocBook source file for the document you are now reading.

[3] http://www.nmt.edu/tcc/help/xml/
[4] http://www.nmt.edu/tcc/help/pubs/xhtml/
[5] http://www.nmt.edu/tcc/help/pubs/xslt/model.xsl
[6] http://www.nmt.edu/tcc/help/pubs/xslt/xslt.xml

2. A brief example

XSLT is not like a programming language: it is not sequentially executed. Instead, an XSLT script is a specification of how the output looks as a function of input. The basic unit is the *template*; a template usually defines how one particular type of XML tag is to be translated.

At the Tech Computer Center, we have installed a Unix program called **`xsltproc`** that will process an XSLT script, transforming a given XML file into an output file in either HTML or XML. See the section below on **`xsltproc`**.

Here is a small example XML file that describes hiking trails inside a park:

```
<!DOCTYPE park SYSTEM "trails.dtd">
<park name="Lincoln Natural Forest">
  <trail dist="3400" climb="medium">Canyon Trail</trail>
  <trail climb="easy" dist="1200">Pickle Madden Trail</trail>
</park>
```

The root element is `<park>`. It contains `<trail>` elements, each describing one hiking trail. We want to translate this to HTML that looks like this:

```
    <html>
      <head>
        <title>Local Hiking Trails</title>
      </head>
      <body>
        <ul>
          <li>Canyon Trail: 3400 feet, climb medium</li>
          <li>Pickle Madden Trail: 1200 feet, climb easy</li>
        </ul>
      </body>
    </html>
```

Here is an XSLT script that does it:

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

<!-- This template processes the root node ("/") -->
<xsl:template match="/">
  <!-- Tags with no xsl: prefix are copied to the output -->
  <html>
    <head>
      <title>Local Hiking Trails</title>
    </head>
    <body>
      <ul>
        <!-- Tags that start with xsl: are instructions on how
          !   to translate the document.  This one says,
          !   translate all the <trail> elements using the
          !   appropriate template (below).
          !-->
        <xsl:apply-templates select="park/trail"/>
```

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

<!-- This template is used to translate the <trail> elements.
 !-->
<xsl:template match="trail">
  <li> <!-- Start a new list item -->
    <!-- Output the text inside the <trail> element-->
    <xsl:value-of select="."/>
    <!-- Output a colon and a space -->
    <xsl:text>: </xsl:text>
    <!-- The next tag outputs the value of the trail element's
     !   "dist=" attribute.
     !-->
    <xsl:value-of select="@dist"/>
    <xsl:text> feet, climb </xsl:text>
    <xsl:value-of select="@climb"/>
  </li> <!-- End of the list item -->
</xsl:template>

</xsl:stylesheet>
```

The first line identifies the file as XML:

```
    <?xml version="1.0"?>
```

The next tag identifies the file as an XSL stylesheet, and gives the URL of the XSLT standard:

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

The next line is an XSLT tag stating that the output should be expressed in HTML and not in XML:

```
    <xsl:output method="html"/>
```

The rest of the file consists of two templates. The first template has the attribute **`match="/"`**, which means that the template applies to the root of the document. The **`"/"`** part is an expression in the XPath language, described in the XPath section below, that selects the root of the document.

The first thing inside this root template is a series of HTML tags like `<html>`, `<head>`, and so on. Because these tag names don't start with **`xsl:`**, they are copied directly to the HTML output file. Note the `<ul>` element that wraps the whole page content in a bullet list; the `<li>` elements inside it will be added elsewhere.

The body of the HTML page, inside the `<body>` element, is created by this line:

```
    <xsl:apply-templates select="park/trail"/>
```

This tag instructs the XSLT processor to go off and find a template that applies to the `<trail>` elements inside the `<park>` at the document's top level. The results of applying that template are inserted at this point in the output file.

After the remainder of the root template, we find the beginning of the second template. This template is used to translate `<trail>` elements:

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

The **match="trail"** attribute uses XPath to select <trail> nodes. As with the other template, the tags inside this template that don't start with **xsl:** are copied to the HTML output file.

The first thing inside this template is the <li> opening tag that will surround the content for this line and make it a bullet in the <ul> bullet list.

The next thing we want to add to the HTML page is the name of the trail. In the XML input, the trail name is the text between the <trail> and </trail> tags. The XPath expression for the content of a node is **"."**, and the <xsl:value-of> tag is used to insert that content into the output file:

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

The next XSLT element outputs a colon and a space to the HTML page we are building:

```
    <xsl:text>: </xsl:text>
```

To output the value of the **dist="..."** attribute, we again use <xsl:value-of>. This time the XPath expression selects an attribute of the context node (which is <trail> inside this template) by using the **@** operator:

```
    <xsl:value-of select="@dist"/>
```

This example will give you the general flavor of XSLT. After covering the XPath expression language used in XSLT (and in other XML-based tools), we'll move on to the actual XSLT tags.

3. Namespaces and XSLT

Since the XSLT stylesheet and the file it operates on are both XML documents, we need some way to distinguish between more than one set of XML element names. Each set of names is called a *namespace*.

The solution to this problem is to define a two-part element name:

```
    <namespace:element-name...>
```

where *namespace* defines which set of element names we're talking about, and *element-name* is the element name inside that namespace.

In an XSLT stylesheet, any element name that doesn't have an **xsl:** prefix represents an element (with its content, if any) that gets written to the output.

Besides the **xsl:** namespace and the namespace of the elements you are writing to the output, you may also use other namespaces. You can control which namespaces are output and which are acted on: see the section on the attributes of the root element, below.

For an example of a situation where you need to refer to other namespaces, see the section on extension elements below.

For more information, see the reference for XML namespaces[7].

[7] http://www.w3.org/TR/1999/REC-xml-names-19990114/

4. XPath reference

The XPath language is used to describe locations in a document tree. It is the expression language used by XSLT. See the XPath specification[8] for complete details.

A well-formed XML document can be visualized as a tree (in the computer science meaning of the term), and this view is used throughout XPath and XSLT. For example, if a certain document represents a book, it might have a top-level tag `<book>`, containing elements like `<chapter>` and `<appendix>`.

A few important definitions:

- A *node* is the basic building block of the document tree, that is, the data structure used to represent an XML document during its processing by an XSLT script.
- The node representing the outermost tag of the document is called the *root node* of the tree.
- All nodes except the root node of the tree have a *parent node*, and many nodes can have *child nodes* under them.

  To continue the example above, the `<book>` node is the root, and its children are the `<chapter>` and `<appendix>` elements.
- All XPath expressions are evaluated in the context of a particular node (location) in the document. That node is called the *context node*.
- The *context size* is the number of children of the context node's parent. For example, if the context node is one of seven children of its parent, the context size is seven.
- The *context position* is the child number of the context node relative to its parent. For example, if the context node is the third of seven children of its parent, its context position is three.

4.1. Data types in XPath

XPath expressions use these data types:

`node-set`
: Many XPath operations select a set of zero or more nodes. For example, the expression **`"address"`** selects all of the `<address>` elements that are children of the context node. This node-set may contain no nodes, one node, or many.

`boolean`
: A true or false value.

`number`
: Numbers in XPath are represented using floating point.

`string`
: A string of characters.

4.2. Node types

These types of nodes are used to represent a document as a tree:

- A *document* node roots the tree. It will always have as its child an element node for the outermost element of the document. It may also have comment or processing instruction nodes as children, if those nodes appear outside the document.

[8] http://www.w3.org/TR/xpath

- Each *element node* represents one XML tag and its children, if any.
- *Attribute nodes* represent attributes of an element. Such nodes have element nodes as a parent, but are not considered ordinary children of the parent element.
- Chunks of text inside an element become *text nodes*.
- Comments in the document are represented as *comment nodes*.
- *Processing instructions* come from XML's **<?...?>** construct.

4.3. Node tests

Most XPath expressions are used to select zero or more nodes from the document. For example, the XPath expression **cue** selects all <cue> child elements of the context node.

These functions are used to select certain special node sets:

text()
: This function selects all the text children of the context node.

comment()
: Selects all comments that are children of the context node.

processing-instruction()
: Selects all children of the context node that are processing instructions.

4.4. Axis selection in XPath

XPath expressions can select nodes using a number of different *axis* specifiers. Each axis describes a different set of nodes relative to the context node, a different direction. For example, using the **child** axis allows you to select only nodes that are children of the context node. The **child** axis is the default axis.

To select nodes from a specific axis, use the syntax ***axis*::*e*** where *e* is any XPath expression. For example, **ancestor::cluster** would select all <cluster> elements that are ancestors of the context node.

Here are the axis specifiers:

child::
: Selects children of the context node. Attribute nodes are not included; use the **attribute::** axis to get attribute nodes.

parent::
: Selects only the parent node, if there is one. Can be abbreviated as "**../**"

self::
: Selects only the context node. This can be abbreviated as "**./**".

attribute::
: Selects only the attributes of the context node. Can be abbreviated as "**@**".

ancestor::
: Refers to all ancestors of the context node: its parent, its parent's parent, and so on up to and including the document node.

ancestor-or-self::
: Refers to the context node and its ancestors.

descendant::
Refers to the descendants of the context node: its children, their children, and so on up to and including the leaves of the tree.

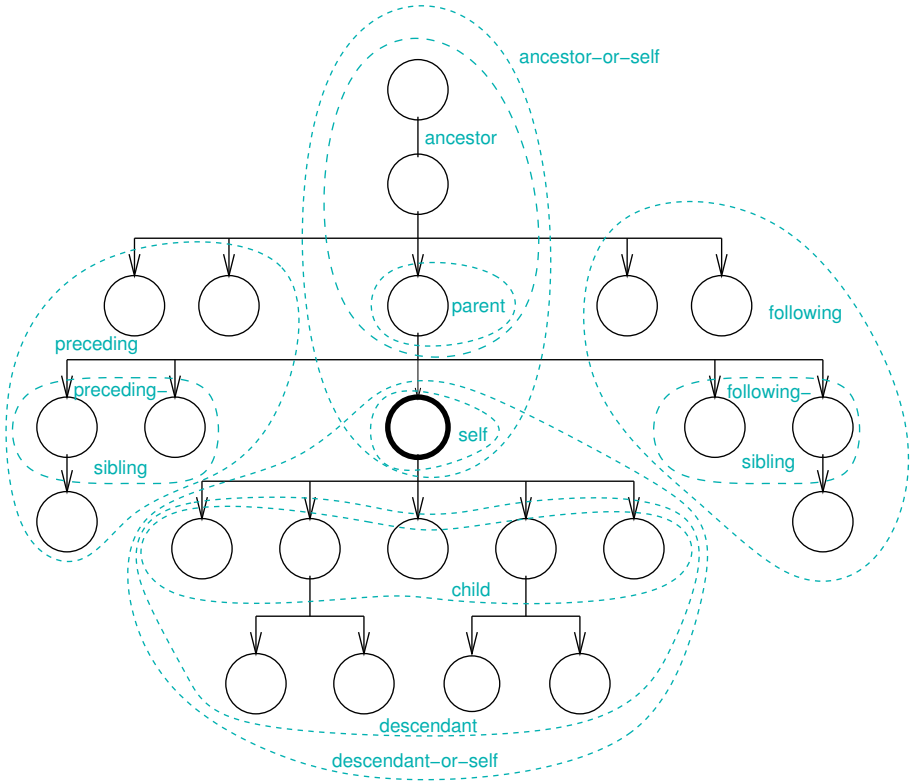
descendant-or-self::
Refers to the context node and its descendants.

preceding-sibling::
Refers to all children of the context node's parent that occur before the context node.

following-sibling::
Refers to all children of the context node's parent that occur after the context node.

preceding::
All nodes that precede the context node in the whole document. This set does not include the context node's descendants or attributes.

following::
All nodes that follow the context node in the whole document. This set does not include the context node's descendants or attributes.



4.5. XPath operators

Here are the operators used in XPath expressions. In the table below, *e* stands for any XPath expression.

e1+e2	If *e1* and *e2* are numbers, their sum.
e1-e2	*e1* minus *e2*.
e1*e2	The product of *e1* and *e2*.

e1* div *e2	If *e1* and *e2* are numbers, their quotient as a floating-point value.
e1* mod *e2	The floating-point remainder of *e1* divided by *e2*.
e1* = *e2	Tests to see if *e1* equals *e2*.
e1* < *e2	Tests to see if *e1* is less than *e2*. You can't say ***e1* < *e2*** inside an attribute: the less-than sign must be escaped as **"<"**.
e1* <= *e2	Tests to see if *e1* is less than or equal to *e2*.
e1* > *e2	Tests for greater-than.
e1* >= *e2	Tests for greater or equal.
e1* != *e2	Tests for inequality.
e1* and *e2	True if both *e1* and *e2* are true. If *e1* is false, *e2* is not evaluated.
e1* or *e2	True if either *e1* or *e2* is true. If *e1* is true, *e2* is not evaluated.
/e	Evaluate *e* starting at the document node. For example, **"/barge"** selects the <barge> element that is the child of the document node.
e1/e2	The / operator separates levels in a tree. For example, **"/barge/load"** selects all <load> children of the <barge> element child of the document node.
//e	Abbreviation for **descendant-or-self::*e***.
./e	Abbreviation for **self::*e***.
../e	Abbreviation for **parent::*e***.
@e	Abbreviation for **attribute::*e***.
e1*|*e2	Selects the union of nodes that match *e1* and those that match *e2*.
*	A wild-card operator; matches all nodes of the proper type for the context. For example, **"*"** selects all child elements of the context node, and **"feet/@*"** selects all attributes of the context node's <feet> children.
***e1*[*e2*]**	Square brackets enclose a *predicate*, which specifies an expression *e2* that selects nodes from a larger set *e1*.For example, in the XPath expression **"para[@class='note']"**, the **para** selects all <para> children of the context node, and then the predicate selects only the children that have an attribute **class="note"**. Another example: **"item[1]"** would select the first <item> child of the context node.
$e	The dollar sign indicates that the following name is a variable name. For example, in an XSLT script, if variable n is set to 357, <xsl:value-of select="$n"/> is expanded to the string **"357"**.

Here is a table showing the precedence of the XPath operators, from highest to lowest:

- **() $**
- ***e*₁[*e*₂]**
- **::**
- **. ..**
- **/ //**
- **|**
- Unary **-**
- *** div mod**
- **+ -**
- **< <= >= >**
- **= !=**
- **and**
- **or**

4.6. XPath functions

XPath supplies these functions:

boolean(*e*)
: Converts object *e* to Boolean type. False values include numeric zero, empty strings, and empty node sets; other values are considered true.

ceiling(*e*)
: Returns the integer closest to infinity that is less than or equal to *e*. Examples: **ceiling(5.9)** returns **6**; **ceiling(-5.9)** returns **-5**.

concat(*e1*, *e2*, ...)
: Concatenates the string values of its arguments and returns the result as a single string.

contains(*s1*, *s2*)
: True if string *s1* contains *s2*.

count(*e*)
: For a node-set *e*, returns the number of nodes in that set.

false()
: Returns the Boolean "false" value.

floor(*e*)
: Returns the integer closest to minus infinity that is greater than or equal to *e*. Examples: **floor(5.9)** returns **5**; **floor(-5.9)** returns **-6**.

id(*e*)
: For a string *e*, this function returns a node-set containing the element whose **id** attribute matches *e*, if there is one.

lang(*s*)
This function tests whether a language code *s* is a substring of the context node's language attribute (from the **xml:lang** attribute). For example, **lang("en")** would return a true value if the context node has an attribute **xml:lang="en-us"**.

last()
Returns the context size; see the section above on context size.

local-name(*n*)
For a node set ***n***, this function returns the local name of the first element, that is, the element name without any namespace. If the argument is omitted, returns the local name of the context node.

For example, if the context node is an **xsl:message** element, **local-name()** will return the string **"message"**.

normalize-space(*s*)
Returns the string *s*, except that all leading and trailing whitespace are removed, and all internal clumps of whitespace are replaced by single spaces.

If the argument is omitted, it operates on the string value of the current node.

not(*e*)
Returns the Boolean complement of the truth value of expression *e*: true if *e* is false, false if it is true.

number(*e*)
Converts an expression *e* to a number. If *e* is not a valid number, you get the special numeric value **NaN** (not a number). If *e* is a Boolean value, you get 1 for true and 0 for false.

position()
Returns the context position; see context position, above.

round(*e*)
Returns the integer closest to the value of expression *e*. Values with a fractional part of 0.5 are rounded towards infinity. Examples: **round(5.1)** returns **5**; **round(5.5)** returns **6**; and **round(-5.5)** returns **-5**.

starts-with(*s1*, *s2*)
True if string *s1* starts with string *s2*.

string(*e*)
Converts *e* to a string.

string-length(*s*)
Returns the length of string *s*.

substring(*s*, *n1*, *n2*)
Returns a substring of *s* starting at position *n1* (counting from 1), and ending *n2* characters later, or at the end of the string, whichever comes first. You can omit the third argument, and it will return the substring starting at position *n1* and going through the end of *s*. For example, **"substring('abcdefgh', 3, 4)"** returns **"cdef"**.

substring-after(*s1*, *s2*)
If *s2* is a substring of *s1*, returns the part of *s1* after the first occurrence of *s2*; otherwise it returns the empty string.

substring-before(*s1*, *s2*)
If *s2* is a substring of *s1*, returns the part of *s1* before the first occurrence of *s2*; otherwise it returns the empty string.

sum(*n*)
For a node-set *n*, this function converts each node to a number, then returns the sum of those numbers.

translate(*s1*, *s2*, *s3*)
The result is a copy of string *s1* with each occurrence of a character from string *s2* replaced with the corresponding character from string *s3*.

If *s3* is shorter than *s2*, this function will delete from its result any characters from *s2* that don't correspond to characters in *s3*.

true()
Returns the Boolean "true" value.

4.7. Attribute value templates

Often the value of an output attribute is not a constant but some function of variables in the stylesheet. In this case, use "`{braces}`" around an XPath expression to produce the desired result.

For example, here is a small template used to wrap some content in an XHTML `span` element.

```
<xsl:template name="shout">
  <xsl:param name="shout-class">shouting</xsl:param>
  <span class="{$shout-class}">
    <xsl:apply-templates select="$content"/>
  </span>
</xsl:template>
```

If the template above is called without parameters, the value of the `class` attribute will be the default value, "`shouting`". If called with an `xsl:with-param` value, the value of the `class` attribute will be the value of that parameter. For example, this call:

```
<xsl:call-template name="shout">
  Danger Will Robinson!
</xsl:call-template>
```

will produce this result:

```
<span class="shouting">Danger Will Robinson!</span>
```

This call, however:

```
<xsl:call-template name="shout">
  <xsl:with-param name="shout-class" select="'screaming'"/>
  Danger Will Robinson!
</xsl:call-template>
```

produces:

```
<span class="screaming">Danger Will Robinson!</span>
```

5. Overall XSLT stylesheet structure

Here is the overall structure of an XSLT stylesheet:

```
<xsl:stylesheet ...>
  top-level-only elements
  templates
</xsl:stylesheet>
```

The root element must be either <xsl:stylesheet> or <xsl:transform>; they are equivalent.

There are certain elements called *top-level-only elements* that must be just inside the root element. These are discussed below in the section on top-level elements.

The rest of the stylesheet consists of any mixture of these elements:

- Basic template elements are for defining templates and creating modular pieces of templates.
- Output elements are used to write things to the output.
- There are several branching elements that provide iteration and optional content.
- See the section on advanced elements for additional useful features.

6. The root element <xsl:stylesheet>

The root element of any XSLT stylesheet must be <xsl:stylesheet>. This tag is described below, followed by the "top level tags" that must be children of <xsl:stylesheet>, that is, they must be outside all <xsl:template> elements.

6.1. <xsl:stylesheet> attributes

Attributes of <xsl:stylesheet> include:

version (required)
Use **version="1.0"**.

xmlns:xsl
Use **xmlns:xsl="http://www.w3.org/1999/XSL/Transform"** to connect your document to this version of the XSLT standard.

extension-element-prefixes
This attribute defines a prefix you will use to invoke extension elements. See extension elements below.

exclude-result-prefixes
Normally, any element whose name doesn't start with **xsl:** is copied to the output. However, if you are using extension elements, you can use this attribute to specify that elements with certain prefixes are to be processed and not copied to the output. See the section on extension elements below for an example.

7. Top-level elements

The elements described in this section can appear only as children of the root <xsl:stylesheet> element. They specify options that affect the entire translation of the input file to the output.

7.1. <xsl:output>: Select output options

This optional tag defines what tag format to use in the output file. The default is XML format. This tag must be a child of the root <xsl:stylesheet> element.

Attributes include:

method
Values include **xml** for XML output; **html** for output as HTML; and **text** if the output is just ordinary text, not tagged.

version
Identifies the version of the output method.

omit-xml-declaration
If the output method is XML, a **<?xml...?>** processing instruction is written to the output file unless you specify **omit-xml-declaration="yes"**. The default is **"no"**. No XML declaration is written for the other output methods.

indent
Use **indent="yes"** to request that XML or HTMl output be indented. Some XSLT processors may not support this option.

encoding
Specifies the character encoding that will appear in the XML processing instruction, if one is generated. Examples: **ISO-8859-1**, **UTF-8**.

doctype-system
Use this attribute to add a **<!DOCTYPE>** declaration to generated HTML or XML.

If you want to generate a **SYSTEM** document type, set this attribute to the URI containing the DTD.

doctype-public
If you want to generate a **<!DOCTYPE>** declaration using a public identifier, set this attribute to the public identifier and also set the **doctype-system** attribute to the URI corresponding to that public identifier.

Here is an example of an <xsl:output> element for generating strict XHTML 1.0:

```
<xsl:output method="xml" encoding="ISO-8859-1" indent="yes"
    doctype-public="-//W3C//DTD XHTML 1.0 Strict//EN"
    doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"/>
```

The file generated with these output options will start like this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE html PUBLIC
    "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

7.2. <xsl:preserve-space>: Preserving white space

Use this element to specify for which input elements white space is preserved. Attributes:

elements (required)
A list of element names separated by spaces.

For example, this element would cause white space to be preserved in <remark> and <warning> elements:

```
<xsl:preserve-space elements="remark warning"/>
```

7.3. <xsl:strip-space>: Removing non-significant white space

By "non-significant white space," we mean white space that occurs between elements. White space that occurs inside text content is considered significant. This tag specifies for which elements non-significant white space is discarded. Attributes:

elements (required)
A list of element names separated by spaces.

For example, this element would cause white space to be deleted in <rant> elements:

```
<xsl:strip-space elements="rant"/>
```

7.4. <xsl:import>: Use templates from another stylesheet

The purpose of this tag is to allow modularity in the design of stylesheets. After using the <xsl:import> element to refers to another XSLT stylesheet, you can use any of the templates in that other stylesheet.

When your stylesheet and the other stylesheet have templates for the same situation, your stylesheet takes precedence. Every template is assigned a *priority*, and the priority of an imported stylesheet is lower than that of the importing stylesheet.

Even in the case that some template *T1* from your stylesheet overrides another template *T2* from an imported stylesheet, template *T1* can use *T2* in its processing. See the <xsl:apply-imports> tag, below.

Attributes:

href (required)
This attribute defines the URI of the stylesheet you are importing.

Example (note that this tag is always empty):

```
<xsl:import href="http://www.nmt.edu/tcc/help/xml/"/>
```

7.5. <xsl:key>: Create an index to optimize input document access

The purpose of the <xsl:key> element is to define an *index* on information in the input file, so that the XSLT processor can retrieve that information more efficiently than by searching the entire file. This is admittedly an advanced feature, but can be very useful for larger projects.

You must give each key a name, using the **name** attribute, that will be used to refer to it later. Two more attributes describe which nodes to index, and what text to use as the index:

match
To define which nodes of the input document are indexed, use attribute **match="*m*"**, where *m* is an XPath expression describing a node set.

use
To define what content strings are indexed, set this attribute to an XPath expression that describes some content relative to the nodes described by the **match** attribute.

For example, suppose you have a document with <river> elements that have a **map** attribute that you'll be referring to elsewhere. You might set up a key called **river-map-key** by placing this element inside your stylesheet:

```
    <xsl:key name="river-map-key" match="river" use="@map"/>
```

7.6. <xsl:decimal-format>: Define a numeric format

This is a top-level element; it can only occur as a child of the root element of your stylesheet. It has two different functions:

- If you provide a **name** attribute, you create a named numeric format that can be used in the **format-number()** function.
- If you don't provide a **name** attribute, the it defines the default format used for all numbers.

Aside from the **name** attribute, the other attributes are not discussed here. They are chiefly useful for non-USA languages. Refer to the XSLT specification[9] for more details.

8. Basic template elements

Use the XSLT elements below to define templates that specify how to process selected input elements.

8.1. <xsl:template>: Define a template

This is the basic tag specifying how to process some set of input. There are two basic types of templates:

- Templates with a **match="..."** attribute are applied to the input nodes that are selected by that attribute.
- Templates with a **name="..."** attribute are called *named templates*. They are invoked by explicitly calling them with the <xsl:apply-templates> element.

  You must supply either a **match** attribute or a **name** attribute.

Attributes of <xsl:template>:

match
: An XPath expression that selects a set of nodes. For example, **match="trail"** specifies a template that matches all <trail> elements.

name
: Instead of a **match** attribute, you can give your template a name and call it using <xsl:call-template>.

  For example, if you have a template that looks like

```
    <xsl:template name="add-links">
```

  then you could call this template using

```
    <xsl:call-template name="add-links"/>
```

  You can also pass values to a named template using <xsl:with-param>.

[9] http://www.w3.org/TR/xslt

mode
Sometimes you want to use the same content in more than one place. For example, chapter titles might appear both at the beginning of a chapter and also in the table of contents.

To do this, define a **mode="*m*"** attribute on the template for each place you use that content. Then use `<xsl:apply-templates mode="m"...>` to process the content with that mode.

8.2. <xsl:variable>: Define a global or local variable

You can define a variable and set its value using the `<xsl:variable>` element. Once you have defined a variable, you can refer to it in any XPath expression by using a dollar sign (**$**) followed by the name of a variable.

There are two kinds of variables:

- A *global* variable's value is available everywhere in your stylesheet. To define a global variable, place the `<xsl:variable>` element as a child of the root stylesheet element.
- The value of a *local* variable is available only inside the element where it is declared.

There are two ways to specify the value of your variable. You can use an XPath expression in the **select** attribute and the variable will take on the value of that expression. You can, instead, place the value in the content of the `<xsl:variable>` element. For example, these two elements have the same effect:

```
<xsl:variable name="child-page" select="'child.html'"/>
<xsl:variable name="child-page">child.html</xsl:variable>
```

Note in the first example above that the value of the **select** attribute has two levels of quotes. This is necessary because in XPath names usually refer to child elements. If you want an attribute to have a string constant as its value, you must provide a second level of quotes inside the quotes that surround the attribute value.

Here are the attributes of the `<xsl:variable>` element:

select
The value you are assigning to this variable.

8.3. <xsl:apply-templates>: Process a node set with appropriate templates

The purpose of this element is to tell the XSLT processor to find the templates that apply to a given set of elements, and apply them. This happens a lot when you are writing the template to process a certain element, and you want to say "now go and process all the children of this element, using whatever templates apply." Attributes:

select
To specify which nodes you want to process, set this attribute's value to an XPath expression that computes a node-set. Each node in the resulting set will be processed. This attribute is optional; the default node-set consists of all the children of the current node.

mode
This attribute is used when you want to process the same input content more than once in more than one way. For example, if you have chapter titles in a document, you'll need to process them twice: once to put the title on the first page of the chapter, and also to build the table of contents.

If you supply a mode attribute with your `<xsl:apply-template>` element, it will only apply templates that have the same **mode** attribute. In the example, you'd write a regular template (with no **mode** attribute) to process the chapter title as part of the chapter body, and you'd write another

template as <xsl:template mode="toc"...> to process the chapter title in the table of contents.

Here are a couple of examples. The most common usage, this means "process all children using the appropriate templates.":

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

And this example processes all <map> children of the current node:

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

8.4. <xsl:include>: Insert another stylesheet

Another tool for modular XSLT programming is the <xsl:include> tag, which effectively means "insert another stylesheet here." There is one attribute:

href
To include another stylesheet, give its URI as the value of this attribute.

There is a difference between importing and including a stylesheet. The <xsl:import> element lets you use templates from another stylesheet, but those templates have a lower priority and can be overridden by your templates. If you use <xsl:include>, though, any templates in the included stylesheet have the same priority as your other templates; it's as if that stylesheet were copied into the middle of your stylesheet.

8.5. <xsl:param>: Define an argument to be passed into a template

The template is the basic module or building block of the XSLT stylesheet. The <xsl:param> element allows you to write a template that can take argument (parameter) values.

Each parameter to a template must have a unique name, supplied via the **name** attribute.

When a template is applied, the values of its parameters are determined by this process:

1. If the calling template supplies a value for the parameter by using an <xsl:with-param>, the parameter will be set to that value.
2. If the calling template does not supply a value, but the <xsl:param> element defines a default value, the parameter is set to that default value.
3. If the <xsl:param> element of the template doesn't supply a default value, the parameter is set to an empty string.

There are two ways to specify the default value of a parameter. You can supply the value as a **select** attribute, or you can put the value inside the content of the <xsl:param> element. So these two constructs work the same:

```
<xsl:param name="color" select="'green'"/>
<xsl:param name="color">green</xsl:param>
```

Attributes:

name (required)
The name of this parameter.

select
The default value for this parameter, if any.

8.6. <xsl:with-param>: Pass an argument to a template

This element is used inside either <xsl:call-template> or <xsl:apply-templates> to pass argument (parameter) values to the called template. The **name** attribute of this element must match one of the parameter names in the called template; see <xsl:param>, above. Attributes:

name (required)
Specifies the name of the parameter to which we're passing the value.

select
Specifies the value we're supplying to the parameter.

If the **select** attribute is omitted, you can supply the value as content; otherwise, the element is empty. These two constructs have the same effect:

```
    <xsl:with-param name="color" select="'green'"/>
    <xsl:with-param name="color">green</xsl:with-param>
```

9. Output instructions

These elements are used to send various things to the output your stylesheet is producing.

9.1. <xsl:text>: Output literal text

To send some text to the output, embed that text in the content of an <xsl:text> element. For example, this would send the text **"not"**, followed by a newline, followed by the text **"insane"**:

```
    <xsl:text>not
insane</xsl:text>
```

There is one optional attribute:

disable-output-escaping
For output to XML or HTML, special characters like **"<"** are translated to their escaped equivalents, such as **"<**. If your <xsl:text> element has attribute **disable-output-escaping="yes"**, however, such characters will be sent as is, untranslated. The default value is **"no"**. This option is ignored for text output.

9.2. <xsl:value-of>: Output the value of an expresssion

To output the string equivalent of some XPath expression, use this element. Attributes:

select (required)
The XPath expression to be evaluated.

disable-output-escaping
See <xsl:text>.

For example, suppose you have defined a variable named **lap-count** and it currently has a value of 32. This element would place the text **"33"** in the output:

```
    <xsl:value-of select="$lap-count+1"/>
```

9.3. <xsl:element>: Output an element

In most cases it's pretty easy to output an element, say something like a <fruit-basket> tag: you just put that tag in your template, and it gets written.

However, sometimes you need to create an element whose name has to be computed using an XPath expression. The <xsl:element> tag allows you to do this. Here are the attributes:

name (required)
Specifies the name of the element you are creating.

namespace
If you want the element to be from a specific namespace, supply this attribute with the namespace as its value.

use-attribute-sets
The name(s) of one or more attribute sets that this element should carry. If there are multiple attribute sets, their names should be separated by spaces.

The content of the <xsl:element> element in your stylesheet, after processing, becomes the content of the generated tag.

Here are some examples. Suppose that XSLT is processing a template that matches <river> elements that have a **salinity** attribute. You want to generate an element whose name is the same as the name of that attribute. You want the content of the generated element to be the value of the node's **flow-rate** attribute. This would do the trick:

```
<xsl:element name="@salinity">
  <xsl:value-of select="@flow-rate"/>
</xsl:element>
```

9.4. <xsl:attribute>: Output an attribute

The purpose of this tag is to add an attribute to an element, especially if the attribute name is something that has to be computed during processing. The attribute is added to whatever element is being output at that point by your template. Here are the attributes:

name (required)
An XPath expression that specifies the attribute name.

namespace
If used, the value of this attribute is prefixed to the attribute name as a namespace.

Here's an example. This generates a <dish> element with an attribute whose namespace is **dl:**, whose attribute name is the value of the variable **dish-tag**, and whose value is the name of the **dish-lang** variable:

```
<dish>
  <xsl:attribute name="$dish-tag" namespace="'dl'">
    <xsl:value-of select="$dish-lang"/>
  </xsl:attribute>
</dish>
```

9.5. <xsl:number>: Output an element number or formatted number

Sometimes you want to number things, like chapters in a book, or section numbers in an outline (1, 1.1, 1.2, and so on). That's what this element is for. It can also be used to format a number derived from an XPath expression.

The way elements are numbered depends on their position in a node-set. For example, if you have an element called <volume-set> whose children are <volume> elements, the template for the <volume-set> element will probably use this construct to format its children:

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

The **select="volume"** XPath expression produces a node-set containing the <volume> children. So, the template that formats the <volume> element will get a node whose position in that set can be determined with the XPath function **position()**. Then you'll need an <xsl:number> construct to convert that position into a number in the output.

This feature has a lot of attributes that let you control the format of the number, and even its value:

value
: If you just want to format the value of an XPath expression, use this attribute with the expression as its value. For example, <xsl:number value="$bat-count"/> would output the value of the **bat-count** variable. If you want to number elements, though, don't use this attribute.

count
: An XPath pattern that selects the nodes that are counted. For example, if you want your <theorem> and <lemma> tags to be counted on the same system, you can specify **count="theorem|lemma"**.

level
: This attribute controls whether we are numbering elements at just one level, or at multiple levels at once (such as an outline that has numbers like "2.3.1.5"). Values are:

single	Only elements at the same level as the current node are counted.
multiple	Any ancestor nodes that match the **count** attribute are included to form a compound number. For example, if the current node is the 4th child of the 1st child of the 3rd child of its ancestor nodes that match the **count** value, the generated number will be **3.1.4**.
any	All the preceding and ancestor nodes that match the **count** expression are counted in a single sequence, and the number output is in that sequence.

format
: The value of this attribute is a model for how the formatted number should look. Values include:

"1"	Use Arabic numerals. You can supply leading zeroes if you like, so a value of **"001"** would fill each number with up to two leading zeroes.
"a"	Use lowercase letters. The sequence is a, b, c, ..., z, aa, ab, ... and so on.
"A"	Use capital letters.
"i"	Use lowercase Roman numerals.
"I"	Use uppercase Roman numerals.

You can also supply trailing punctuation, so for example **format="1: "** would follow each number with a colon and space.

There are some other attributes, not discussed here, that are useful for non-USA usages. Refer to the XSLT specification[10] for more details.

10. Branching elements

The elements described in this section all you to iterate over a set of nodes, and also to provide processing that happens only in certain cases.

10.1. <xsl:for-each>: Iterate over a set of nodes

To do the same thing to each node in a node set, use the <xsl:for-each> construct. Stuff inside this element gets done once for each element in the node-set. There is one attribute:

select (required)
An XPath expression that specifies the node-set over which you want to iterate.

For example, suppose you are writing a template with child elements of types <daughter> and <son>. This element would process all the children of either type:

```
<xsl:for-each select="daughter|son">...</xsl:for-each>
```

10.2. <xsl:if>: Conditional processing

Sometimes you want to perform an action only when a certain condition is met. The <xsl:if> tag does this conditional processing. This is XSLT's single-branch "if" construct; there is no two-branch form. If you want to perform one thing if a condition is true, and a different thing if that condition is false, you'll have to use the <xsl:choose> construct. There is one required attribute:

test
This XPath expression is evaluated. If it is true, the content of the <xsl:if> element is processed, otherwise that content is not processed.

Here's an example. This writes the value of variable **head-count** to the output, but only if it is an odd number:

```
<xsl:if test="($head-count mod 2)=1">
  <xsl:value-of select="$head-count"/>
</xsl:if>
```

10.3. <xsl:choose>: The multiple-case construct

To produce output with two or more different cases, use the <xsl:choose> construct. It has one <xsl:when> element for each case, plus an optional <xsl:otherwise> element at the end for the "none of the above" case. Each <xsl:when> element has a **test** attribute whose value is an XPath expression; the first one that evaluates to a true value is processed.

Here's an example that outputs the content of the <turtle>children of the current node, and puts the first one in parentheses and the last one in square brackets.

[10] http://www.w3.org/TR/xslt

```
<xsl:for-each select="turtle">
  <xsl:choose>
    <xsl:when test="position() = 1">
      <xsl:text>(</xsl:text>
      <xsl:apply-templates select="."/>
      <xsl:text>)</xsl:text>
    </xsl:when>
    <xsl:when test="position() = count()">
      <xsl:text>[</xsl:text>
      <xsl:apply-templates select="."/>
      <xsl:text>]</xsl:text>
    </xsl:when>
    <xsl:otherwise>
      <xsl:apply-templates select="."/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:for-each>
```

10.4. <xsl:call-template>: Invoke another template

You can package up bits of XSLT in named templates, and call those templates with the <xsl:call-template> construct. Attributes:

name (required)
The name of the template you are calling.

If you want to pass arguments (parameters) to the template, use one <xsl:with-param> element for each value you want to pass.

11. Advanced elements

The elements in this section round out the set of XSLT elements. These elements are used less often but may be helpful in certain cases.

11.1. <xsl:apply-imports>: Use an overridden template

You can use templates that you have imported using the <xsl:import> tag, but you can also override them by providing your own template. The **<xsl:apply-imports/>** tag allows you to use an imported template even if you have overridden it.

There are no attributes and no content. The template you are invoking is the one overridden by the current template.

11.2. <xsl:attribute-set>: Define a named attribute set

Often you will want to add one or more attributes to an element you are generating with the <xsl:element> tag. The <xsl:attribute-set> element lets you create a named set of attributes so you can add them merely by referring to that name with the **use-attribute-sets** attribute of <xsl:element>.

Use one or more <xsl:attribute> children as the content of this element. Those attributes and their values make up the attributes in the set.

Attributes of <xsl:attribute-set> include:

name (required)
Gives this attribute set a name.

use-attribute-sets
Any attribute sets named in this value become part of the attribute set you are creating.

11.3. <xsl:comment>: Output a comment

To write a comment to the output, use this element. The content of the element becomes the content of the comment. There are no attributes. For example, this element in a template:

```
<xsl:comment>Do not edit this file; it is generated
  automatically by an XSLT script.</xsl:comment>
```

would generate this comment in the output:

```
<!--Do not edit this file; it is generated
  automatically by an XSLT script.-->
```

11.4. <xsl:copy>: Shallow copying

The current node is copied in the corresponding location of the output, but its children, and any attributes it may have, are not copied. There is one optional attribute:

use-attribute-sets
You can attach attributes to the new copy of the node by specifying a value for this attribute the names of one or more named attribute sets. See the section on named attribute sets.

11.5. <xsl:copy-of>: Deep copying

To copy an entire subtree of the input document, use this tag with a **select** attribute that describes a node-set. The nodes in that node-set will be added to the output, ordered in the same way they are in the input document. Also, any attributes of those nodes, and their children and all descendants, are recursively copied in the same way.

11.6. <xsl:fallback>: What to do if an extension is missing

If you are trying to use an XSLT extensions (see extension elements, below), and the extension can't be found, you can provide an <xsl:fallback> element to specify what actions should be taken. For example, this fragment shows a reference to an extension element named <exsl:document>. If that element is not available, it writes a message and terminates:

```
<exsl:document href="page.html">
  <xsl:fallback>
    <xsl:message terminate="yes">
      Aieeee! The exsl:document package is missing!
      I'm out of here!
    </xsl:message>
  </xsl:fallback>
</exsl:document>
```

11.7. <xsl:message>: Write a debugging message

This element writes its content to the standard output stream (not to the output document). For an example, see <xsl:fallback> above. Attributes:

terminate
If you use **terminate="yes"**, the XSLT processor will stop execution after writing the message.

11.8. <xsl:namespace-alias>: Assign a prefix to a namespace

This is a pretty obscure feature. It has only one use: writing an XSLT script that generates another XSLT script.

Since normally any tag whose name starts with **xsl:** is processed, we have to provide a way to output a tag in the **xsl:** namespace without processing it. The way we do this involves two steps:

1. You must declare two different prefixes for the **xsl:** namespace in the attributes of the <xsl:stylesheet> tag. One prefix is declared as **xsl:**; you will use this namespace for the elements that you want to process as part of your stylesheet. The other prefix is used to refer to the elements that you want to write to the output document.
2. You must provide a top-level <xsl:namespace-alias> element with a **stylesheet-prefix** attribute whose value is the other prefix (besides **xsl:**), and whose **result-prefix** attribute is **"xsl"**.

For example, suppose your stylesheet starts this way:

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

  <xsl:output method="xml"/>
  <xsl:namespace-alias stylesheet-prefix="out-xsl"
    result-prefix="xsl"/>
  ...
```

then elements in your stylesheet with namespace **xsl:** will be processed, and elements with namespace **out-xsl:** will be written to the output document.

11.9. <xsl:processing-instruction>: Output a processing instruction

Use this element to output a processing instruction (**<?*target properties*?>**). Use the **name** attribute to generate the *target* portion; the content of the element becomes the *properties* portion.

11.10. <xsl:sort>: Process nodes in a given order

This element is used just inside the start of either an <xsl:apply-templates> element or a <xsl:for-each> element. It has the effect of sorting the nodes in the parent element's node-set, so that the content is presented sorted according to some aspect of the content.

You can use multiple <xsl:sort> elements to specify multiple sort keys. That is, if there are two <xsl:sort> elements, the first one specifies the primary key, and the second one the secondary sort key (to be used in comparing items that have the same value for the primary key).

The sort is *stable*, that is, if two items are equal in all key values, they will be in the same order as they were in the original document.

Here are the attributes:

`select`
: An XPath expression that selects the nodes to be sorted. The default value is the node-set of the parent element.

`data-type`
: Use a value of **`"text"`** to treat the items as text strings, or a value of **`"number"`** to treat them as numbers. Default is **`"text"`**.

`order`
: Use **`"asc"`** for ascending order (this is the default), or **`"desc"`** for descending order.

`case-order`
: Specifies how to compare uppercase and lowercase letters. A value of **`"upper-first"`** forces uppercase letters before lowercase ones; the opposite is **`"lower-first"`**.

  The default is language-dependent.

`lang`
: Defines the language to be used in sorting. See the definition of the language codes[11] online.

Here's an example. Suppose you have `<person>` elements that have children named `<surname>` (containing the person's last name) and `<first-name>` (containing the person's given name). You want to process them in order by last name, using the first name as a tiebreaker. Here is a stylesheet fragment that does this:

```
<xsl:for-each select="person">
  <xsl:sort select="surname"/>
  <xsl:sort select="first-name"/>
  <xsl:apply-templates select="."/>
</xsl:for-each>
```

12. XSLT functions

In addition to the XPath functions defined above, XSLT supplies a number of additional functions.

12.1. `current()`: Return the current node

The **`current()`** function returns a node-set containing only the current (context) node. In most situations, this is exactly the same result as the XPath expression **`"."`**, but there is one case with a very important difference.

Recall that XPath expressions can have several levels of selection. For example, the XPath expression **`"park/tree"`** selects `<park>` children of the context node, and then it selects `<tree>` children of *those* nodes. So at each stage of evaluation of that XPath expression, **`"."`** has a different meaning: at each stage it means the set of nodes selected so far.

The critical difference in the **`current()`** function is that it always refers to the node that was the context node before any of the XPath expression was evaluated.

[11] http://www.ietf.org/rfc/rfc1766.txt

12.2. `document()`: Pull in content from other documents

There is one optional argument:

```
    document(uri)
```

where *`uri`* is the URI of a document. The return value is a node-set containing that document as a tree. If the argument is an empty string (**`'document("")'`**), you get back your own stylesheet (the one in which the **`document()`** call occurs) as a tree. This allows you to access content in another document (or your stylesheet) the same way you'd access the content of the input document: write template rules for that content and invoke it with `<xsl:apply-templates>` or `<xsl:call-template>`.

12.3. `format-number()`: Convert a number to a string

This function takes two required arguments and a third optional argument:

```
    format-number(n, p [,f])
```

where:

`n`	is the number to be formatted
`p`	is a format pattern string using the characters described below
`f`	if supplied, this is the name of a named `<xsl:decimal>` format item that will be used to determine internationalization features of the formatting, such as the grouping separator.

Here are the characters used in the formatting pattern ***`p`***:

Table 1. Format pattern characters for `format-number()`

`#`	Denotes a digit. Leading and trailing zeroes (and a trailing decimal, if any) will disappear.
`0`	Denotes a digit, but a digit always appears, even if it is a zero.
`-`	Shows the position of the minus sign.
`.`	Shows the position of the decimal point.
`,`	Positions the grouping separator for thousands. For example, **`"##,###.00"`**.
`%`	Multiplies the number by one hundred and displays it as a percentage.
`;`	You can supply two patterns separated by a semicolon; the first one will be used for positive numbers and the second for negative numbers.
other	Literal characters are carried through to the result.

12.4. `generate-id()`: Generate a unique identifier

The purpose of this function is to generate a string of characters that uniquely identifies a node. Such values are useful for attributes of type ID.

The function takes one optional argument, a node-set:

```
    generate-id(S)
```

where *S* is a node-set. If *S* is not given, the function generates an identifier for the context node. If *S* is empty, the function returns an empty string. If *S* contains more than one node, the function operates on the one that occurs first in the document.

Within a given execution, this function will always produce the same value for a given node. There is no guarantee that it will produce the same value on a different execution of the stylesheet.

12.5. key(): Refer to an index entry

This function is used to retrieve a set of nodes from anywhere in the document, using the index specified elsewhere by an `<xsl:key>` element.

The calling sequence is:

```
key(keyName, keyValue )
```

where *keyName* matches the **name** attribute of an `<xsl:key>` element, and *keyValue* is a string. The result is a node-set containing all the nodes whose value for that key matches *keyValue*.

Here's an example. Suppose we declare a key like this:

```
<xsl:key name="river-map-key" match="river" use="@map"/>
```

Then we might process a set of `<river>` elements that have an attribute **map="Elfego"** with this query:

```
<xsl:apply-templates select="key('river-map-key', 'Elfego')"/>
```

12.6. system-property(): Return a system property value

The purpose of this function is to interrogate certain properties of the XSLT processor. The calling sequence is:

```
system-property(name)
```

where *name* is the name of a property. The function returns the value of the named property. In addition to any names that the XSLT processor may support, all processors must return values for these property names:

xsl:version
A number representing the supported version of XSLT. At the moment, it will probably return 1.0.

xsl:vendor
The name of the vendor that built the XSLT processor.

xsl:vendor-url
The URL of the vendor's homepage.

13. Built-in templates

XSLT supplies several built-in, default templates. These templates make it unnecessary to write templates for every possible situation; you need write templates only for the specific nodes you want to process.

The first built-in template operates on any document node (/) or any element node (*) that doesn't have a more specific template. It uses `xsl:apply-templates` so that its child elements will be processed.

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

If you are using modes, the equivalent of this template also operates for any document or element node. For example, if you use a mode called "`toc`" somewhere, XSLT will supply a template that works like this:

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

Another built-in template operates for any text or attribute nodes without more specific templates. It has the effect of copying any text to the output by default.

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

Finally, this built-in template has the effect of ignoring comments and processing instructions.

```
<xsl:template match="processing-instruction()|comment()"/>
```

14. Extension elements

You may need functions that are not part of standard XSLT. A number of organizations have defined so-called *extension elements* that provide these additional functions.

To use an extension, you must add two attributes to the `<xslt:stylesheet>` element of your stylesheet:

1. Declare a namespace for the extension elements by adding an attribute of the form **`xmlns:`*`n`*`="`*`u`*`"`**, where *n* is the namespace you are declaring, and *u* is the URI of the extension element's definition.
2. Tell XSLT to process this namespace, instead of writing it to the output, by using an attribute of the form **`extension-element-prefixes="`*`n`*`"`**, where *n* is the same namespace name used in the previous step.

We'll just give one example; more extensions will arise as XSLT evolves.

14.1. The `exsl:document` extension

With stock XSLT, all output goes to one place: the output document. However, an organization called EXSLT has published an extension element that allows you to sent output to other files. See the EXSLT homepage[12] for more information.

Here is an example of an `<xsl:stylesheet>` element that includes the attributes necessary to use the **`exsl:document`** extension:

```
    <xsl:stylesheet version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xmlns:exsl="http://exslt.org/common"
      extension-element-prefixes="exsl">
```

[12] http://www.exslt.org/howto.html

Once you have done this, in order to write some content to a file named *F*, embed that content in an element that looks like this:

```
<exsl:document href="F">
  ...
</exsl:document>
```

15. Using the `xsltproc` processor

Before you try to apply your XSLT script, you may want to validate the XML file against its DTD. To do this, use the **`xmllint`** program:

```
xmllint --valid --noout filename.xsl
```

where *`filename`*`.xsl` is the name of your XML file. Make sure your DTD is in the same directory and named in a **`<!DOCTYPE ...>`** declaration.

For example, if your XML document uses a DTD named **`trails.dtd`** and its root element is `<trails>`, its first line should look like this:

```
<!DOCTYPE trails SYSTEM "trails.dtd">
```

To transform a file using XSLT, use **`xsltproc`**:

```
xsltproc -o output.html stylesheet.xsl file.xml
```

where ***`stylesheet`*`.xsl`** is the XSLT stylesheet, ***`file`*`.xml`** is the XML input file, and ***`output`*`.html`** is the output file to be written.