SAXON home page

XSL Elements

This page lists the standard XSL elements, all of which are supported in SAXON Stylesheets. For extension elements provided with the SAXON product, see extensions.html.

SAXON implements the XSLT version 1.0 specification from the World Wide Web Consortium: see Conformance. This page is designed to give a summary of the features: for the full specification, consult the official standard.

Contents

Standard XSLT Elements

xsl:apply-imports

The **xsl:apply-imports** element is used in conjunction with imported stylesheets. There are no attributes. The element may contain zero of more xsl:with-param elements (as permitted in XSLT 1.1).

At run-time, there must be a *current template*. A current template is established when a template is activated as a result of a call on xsl:apply-templates. Calling xsl:call-template does not change the current template. Calling xsl:for-each does not (as the XSLT standard says it should) cause the current template to become null.

The effect is to search for a template that matches the current node and that is defined in a stylesheet that was imported (directly or indirectly, possibly via xsl:include) from the stylesheet containing the current template, and whose mode matches the current mode. If there is such a template, it is activated using the current node. If not, the call on xsl:apply-imports has no effect.

It is not possible to supply parameters to a template invoked using xsl:apply-imports.

---

xsl:apply-templates

The xsl:apply-templates element causes navigation from the current element, usually but not necessarily to process its children. Each selected node is processed using the best-match **xsl:template** defined for that node.

The **xsl:apply-templates** element takes an optional attribute, **mode**, which identifies the processing mode. If this attribute is present, only templates with a matching **mode** parameter will be considered when searching for the rule to apply to the selected elements.

It also takes an optional attribute, **select**.

If the **select** attribute is *omitted*, apply-templates causes all the immediate children of the current node to be processed: that is, child elements and character content, in the order in which it appears. Character content must be processed by a template whose match pattern will be something like "*/text()". Child elements similarly are processed using the appropriate template, selected according to the rules given below under xsl:template.

If the **select** attribute is *included*, it must be a *node set expression* which identifies the nodes to be processed. All nodes selected by the expression are processed.

The **xsl:apply-templates** element is usually empty, in which case the selected nodes are processed in the order they appear in the source document. However it may include xsl:sort and/or xsl:param elements:

- For sorted processing, one or more child xsl:sort elements may be included. These define the sort order to be applied to the selection. The sort keys are listed in major-to-minor order.
- To supply parameters to the called template, one or more xsl:with-param elements may be included. The values of these parameters are available to the called template.

The selected nodes are processed in a particular *context*. This context includes:

- A current node: the node being processed
- A current node list: the list of nodes being processed, in the order they are processed (this affects the value of the position() and last() functions)
- A set of variables, which initially is those variable defined as parameters

Some examples of the most useful forms of select expression are listed below:

Expression	Meaning
XXX	Process all immediate child elements with tag XXX
*	Process all immediate child elements (but not character data within t element)
../TITLE	Process the TITLE children of the parent element
XXX[@AAA]	Process all XXX child elements having an attribute named AAA
@*	Process all attributes of the current element
*/ZZZ	Process all grandchild ZZZ elements
XXX[ZZZ]	Process all child XXX elements that have a child ZZZ
XXX[@WIDTH and not(@WIDTH="20")]	Process all child XXX elements that have a WIDTH attribute whose value is not "20"
AUTHOR[1]	Process the first child AUTHOR element
APPENDIX[@NUMBER] [last()]	Process the last child APPENDIX element having a NUMBER attribu
APPENDIX[last()] [@NUMBER]	Process the last child APPENDIX element provided it has a NUMBEI attribute

The full syntax of select expressions is given in XPath Expression Syntax

---

xsl:attribute

The **xsl:attribute** element is used to add an attribute value to an **xsl:element** element or

general formatting element, or to an element created using **xsl:copy**. The attribute must be output immediately after the element, with no intervening character data. The name of the attribute is indicated by the **name** attribute and the value by the content of the xsl:attribute element.

The attribute name is interpreted as an *attribute value template*, so it may contain string expressions within curly braces. The full syntax of string expressions is given in XPath Expression Syntax

For example, the following code creates a <FONT> element with several attributes:

```
<xsl:element name="FONT">
    <xsl:attribute name="SIZE">4</xsl:attribute>
    <xsl:attribute name="FACE">Courier New</xsl:attribute>
Some output text
</xsl:element>
```

There are two main uses for the **xsl:attribute** element:

- It is the only way to set attributes on an element generated dynamically using xsl:element
- It allows attributes of a literal result element to be calculated using xsl:value-of.

The xsl:attribute must be output immediately after the relevant element is generated: there must be no intervening character data (other than white space which is ignored). SAXON outputs the closing ">" of the element start tag as soon as something other than an attribute is written to the output stream, and rejects an attempt to output an attribute if there is no currently-open start tag. Any special characters within the attribute value will automatically be escaped (for example, "<" will be output as "<")

If two attributes are output with the same name, the second one takes precedence.

Saxon permits the additional attribute **saxon:disable-output-escaping**. If this is set to the value "yes", then the attribute value will be output as-is, without escaping of special characters. This affects both the normal XML escaping (e.g. of ampersand) and the special URL escaping that occurs with non-ASCII characters in HTML URL attributes (e.g. href) which normally causes a space to be output as %20.

---

xsl:attribute-set

The **xsl:attribute-set** element is used to declare a named collection of attributes, which will often be used together to define an output style. It is declared at the top level (subordinate to xsl:stylesheet).

An attribute-set contains a collection of xsl:attribute elements.

The attributes in an attribute-set can be used in several ways:

- They can be added to a literal result element by specifying xsl:use-attribute-sets in the list of attributes for the element. The value is a space-separated list of attribute-set names. Attributes specified explicitly on the literal result element, or added using xsl:attribute, override any that are specified in the attribute-set definition.
- They can be added to an element created using xsl:element, by specifying use-attribute-sets in the list of attributes for the xsl:element element. The value is a space-separated list of attribute-set names. Attributes specified explicitly on the literal result element, or added using xsl:attribute, override any that are specified in the attribute-set definition.
- One attribute set can be based on another by specifying use-attribute-sets in the list of attributes for the xsl:attribute-set element. Again, attributes defined explicitly in the attribute set override any that are included implicitly from another attribute set.

Attribute sets named in the xsl:use-attribute-sets or use-attribute-sets attribute are applied in

the order given: if the same attribute is generated more than once, the later value always takes precedence.

---

xsl:call-template

The **xsl:call-template** element is used to invoke a named template.

The **name** attribute is mandatory and must match the name defined on an xsl:template element.

Saxon supports an additional attribute **saxon:allow-avt**. If this is present and is set to the value "yes", then the **name** attribute may be written as an attribute value template, allowing the called template to be decided at run-time. The string result of evaluating the attribute value template must be a valid QName that identifies a named template somewhere in the stylesheet.

Parameters to the called template may be defined using xsl:with-param elements nested within the xsl:call-template element.

The context of the called template (for example the current node and current node list) is the same as that for the calling template; however the variables defined in the calling template are not accessible in the called template.

---

xsl:choose

The **xsl:choose** element is used to choose one of a number of alternative outputs. The element typically contains a number of xsl:when elements, each with a separate test condition. The first xsl:when element whose condition matches the current element in the source document is expanded, the others are ignored. If none of the conditions is satisfied, the xsl:otherwise child element, if any, is expanded.

The test condition in the xsl:when element is a boolean expression. The full syntax of expressions is given in XPath Expression Syntax

Example:

```
<xsl:choose>
    <xsl:when test="@cat='F'">Fiction</xsl:when>
    <xsl:when test="@cat='C'">Crime</xsl:when>
    <xsl:when test="@cat='R'">Reference</xsl:when>
    <xsl:otherwise>General</xsl:otherwise>
</xsl:choose>
```

---

xsl:comment

The **xsl:comment** element can appear anywhere within an xsl:template. It indicates text that is to be output to the current output stream in the form of an XML or HTML comment.

For example, the text below inserts some JavaScript into a generated HTML document:

```
<script language="JavaScript">
    <xsl:comment>
        function bk(n) {
            parent.frames['content'].location="chap" + n + ".1.html";
        }
    //</xsl:comment>
</script>
```

Note that special characters occurring within the comment text will *not* be escaped.

The xsl:comment element will normally contain text only but it may contain other elements such as **xsl:if** or **xsl:value-of**. However, it should not contain literal result elements.

Tip: the xsl:comment element can be very useful for debugging your stylesheet. Use comments in the generated output as a way of tracking which rules in the stylesheet were invoked to produce the output.

---

xsl:copy

The **xsl:copy** element causes the current XML node in the source document to be copied to the output. The actual effect depends on whether the node is an element, an attribute, or a text node.

For an element, the start and end element tags are copied; the attributes, character content and child elements are copied only if **xsl:apply-templates** is used within xsl:copy.

Attributes of the generated element can be defined by reference to a named attribute set. The optional use-attribute-sets attribute contains a white-space-separated list of attribute set names. They are applied in the order given: if the same attribute is generated more than once, the later value always takes precedence.

The following example is a template that copies the input element to the output, together with all its child elements, character content, and attributes:

```
<xsl:template match="*|text()|@*">
    <xsl:copy>
        <xsl:apply-templates select="@*"/>
        <xsl:apply-templates/>
    </xsl:copy>
</xsl:template>
```

---

xsl:copy-of

The xsl:copy-of element copies of the value of the expression in the mandatory **select** attribute to the result tree

If this expression is a string, a number, or a boolean, the effect is the same as using xsl:value-of. It is usually used where the value is a nodeset or a result tree fragment.

---

xsl:decimal-format

The **xsl:decimal-format** element is used at the top level of a stylesheet to indicate a set of localisation parameters. If the xsl:decimal-format element has a name attribute, it identifies a named format; if not, it identifies the default format.

In practice decimal formats are used only for formatting numbers using the format-number() function in XSL expressions. For details of the attributes available, see the XSLT specification.

---

xsl:document

The xsl:document element is new in XSLT 1.1, and replaces the previous extension element saxon:output. It is used to direct output to a secondary output destination.

Most of the attributes are the same as for xsl:output, including the additional extension attributes supported by Saxon. The one addition is the mandatory href attribute, which defines the destination of the output after serialization.

Here is an example that uses xsl:document:

```
<xsl:template match="preface">
    <xsl:document href="{$dir}\preface.html">
        <html><body bgcolor="#00eeee"><center>
            <xsl:apply-templates/>
        </center><hr/></body></html>
    </xsl:document>
    <a href="{$dir}\preface.html">Preface</a>
</xsl:template>
```

Here the body of the preface is directed to a file called preface.html (prefixed by a constant that supplies the directory name). Output then reverts to the previous destination, where an HTML hyperlink to the newly created file is inserted.

---

xsl:element

The **xsl:element** is used to create an output element whose name might be calculated at run-time.

The element has a mandatory attribute, **name**, which is the name of the generated element. The name attribute is an *attribute value template*, so it may contain string expressions inside curly braces.

The attributes of the generated element are defined by subsequent **xsl:attribute** elements. The content of the generated element is whatever is generated between the **<xsl:element>** and **</xsl:element>** tags.

Additionally, attributes of the generated element can be defined by reference to a named attribute set. The optional use-attribute-sets attribute contains a white-space-separated list of attribute set names. They are applied in the order given: if the same attribute is generated more than once, the later value always takes precedence.

For example, the following code creates a <FONT> element with several attributes:

```
<xsl:element name="FONT">
    <xsl:attribute name="SIZE">4</xsl:attribute>
    <xsl:attribute name="FACE">Courier New</xsl:attribute>
Some output text
</xsl:element>
```

---

xsl:fallback

The **xsl:fallback** element is used to define recovery action to be taken when an instruction element is used in the stylesheet and no implentation of that element is available. An element is an instruction element if its namespace URI is the standard URI for XSL elements or if its namespace is identified in the xsl:extension-element-prefixes attribute of a containing literal result element, or in the extension-element-prefixes attribute of the xsl:stylesheet element.

If the xsl:fallback element appears in any other context, it is ignored, together with all its child and descendant elements.

There are no attributes.

If the parent element can be instantiated and processed, the xsl:fallback element and its descendants are ignored. If the parent element is not recognised of if any failure occurs instantiating it, all its xsl:fallback children are processed in turn. If there are no xsl:fallback children, an error is reported.

---

xsl:for-each

The **xsl:for-each** element causes iteration over the nodes selected by a node-set expression.

It can be used as an alternative to **xsl:apply-templates** where the child nodes of the current node are known in advance. There is a mandatory attribute, **select**, which defines the nodes over which the statement will iterate. The XSL statements subordinate to the xsl:for-each element are applied to each source node seleced by the node-set expression in turn.

The full syntax of node-set expressions is given in XPath Expression Syntax

The xsl:for-each element may have one or more **xsl:sort** child elements to define the order of sorting. The sort keys are specified in major-to-minor order.

The expression used for sorting can be any string expressions. The following are particularly useful:

- element-name, e.g. TITLE: sorts on the value of a child element
- attribute-name, e.g. @CODE: sorts on the value of an attribute
- ".": sorts on the character content of the element
- "qname(.)": sorts on the name of the element

Example 1:

```
<xsl:template match="BOOKLIST">
    <TABLE>
    <xsl:for-each select="BOOK">
        <TR>
        <TD><xsl:value-of select="TITLE"/></TD>
        <TD><xsl:value-of select="AUTHOR"/></TD>
        <TD><xsl:value-of select="ISBN"/></TD>
        </TR>
    </xsl:for-each>
    </TABLE>
</xsl:template>
```

Example 2: sorting with xsl:for-each. This example also shows a template for a BOOKLIST element which processes all the child BOOK elements in order of their child AUTHOR elements.

```
<xsl:template match="BOOKLIST">
    <h2>
        <xsl:for-each select="BOOK">
            <xsl:sort select="AUTHOR"/>
            <p>AUTHOR: <xsl:value-of select="AUTHOR"/></p>
            <p>TITLE: <xsl:value-of select="TITLE"/></p>
            <hr/>
        </xsl:for-each>
    </h2>
</xsl:template>
```

xsl:if

The **xsl:if** element is used for conditional processing. It takes a mandatory **test** attribute, whose value is a boolean expression. The contents of the xsl:if element are expanded only of the expression is true.

The full syntax of boolean expressions is given in XPath Expression Syntax

Example:

```
<xsl:if test="@preface">
        <a href="preface.html">Preface</a>
</xsl:if>
```

This includes a hyperlink in the output only if the current element has a **preface** attribute.

xsl:include

The **xsl:include** element is always used at the top level of the stylesheet. It has a mandatory **href** attribute, which is a URL (absolute or relative) of another stylesheet to be textually included within this one. The top-level elements of the included stylesheet effectively replace the xsl:include element.

xsl:include may also be used at the top level of the included stylesheet, and so on recursively.

xsl:import

The **xsl:import** element is always used at the top level of the stylesheet, and it must appear before all other elements at the top level. It has a mandatory **href** attribute, which is a URL (absolute or relative) of another stylesheet to be textually included within this one. The top-level elements of the included stylesheet effectively replace the xsl:import element.

xsl:import may also be used at the top level of the included stylesheet, and so on recursively.

The elements in the imported stylesheet have lower precedence than the elements in the importing stylesheet. The main effect of this is on selection of a template when xsl:apply-templates is used: if there is a matching template with precedence X, all templates with precedence less than X are ignored, regardless of their priority.

xsl:key

The **xsl:key** element is used at the top level of the stylesheet to declare an attribute, or other value, that may be used as a key to identify nodes using the key() function within an expression. Each xsl:key definition declares a named key, which must match the name of the key used in the key() function.

The set of nodes to which the key applies is defined by a pattern in the match attribute: for example, if match="ACT|SCENE" then every ACT element and every SCENE element is indexed by this key.

The value of the key, for each of these matched elements, is determined by the **use** attribute. This is an expression, which is evaluated for each matched element. If the expression returns a node-set, the string value of each node in this node-set acts as a key value. For example, if use="AUTHOR", then each AUTHOR child of the matched element supplies one key value. If the expression returns any other value, the value is converted to a string and that string acts as the key.

Note that

1. Keys are not unique: the same value may identify many different nodes
2. Keys are multi-valued: each matched node may have several (zero or more) values of the key, any one of which may be used to locate that node
3. Keys can only be used to identify nodes within a single XML document: the key() function will return nodes that are in the same document as the current node.

All three attributes, name, match, and use, are mandatory.

xsl:message

The **xsl:message** element causes a message to be displayed. The message is the contents of the xsl:message element.

There is an optional attribute **terminate** with permitted values yes and no; the default is no. If the value is set to yes, processing of the stylesheet is terminated after issuing the message.

By default the message is displayed on the standard error output stream. You can supply your own message Emitter if you want it handled differently. This must be a class that implements the com.icl.saxon.output.Emitter interface. The content of the message is in general an XML

fragment. You can supply the emitter using the -m option on the command line, or the setMessageEmitter() method of the Controller class.

No newline is added to the message; if you want one, include it in the text using
, as in the example below.

Example: This example displays an error message.

```
<xsl:template match="BOOK">
    <xsl:if test="not(@AUTHOR)">
        <xsl:message>Error: BOOK found with no AUTHOR!&#xa;</xsl:m
    </xsl:if>
...
</xsl:template>
```

xsl:namespace-alias

The **xsl:namespace-alias** element is a top-level element that is used to control the mapping between a namespace URI used in the stylesheet and the corresponding namespace URI used in the result document.

Normally when a literal result element is encountered in a template, the namespace used for the element name and attribute names in the result document is the same as the namespace used in the stylesheet. If a different namespace is wanted (e.g. because the result document is a stylesheet using the XSLT namespace), then xsl:namespace-alias can be used to define the mapping.

Example: This example allows the prefix outxsl to be used for output elements that are to be associated with the XSLT namespace. It assumes that both namespaces xsl and outxsl have been declared and are in scope.

```
<xsl:namespace stylesheet-prefix="outxsl" result-prefix="xsl"/>
```

xsl:number

The **xsl:number** element outputs the sequential number of a node in the source document. It takes an attribute **count** whose value is a pattern indicating which nodes to count; the default is to match all nodes of the same type and name as the current node.

The **level** attribute may take three values: "single", "any", or "multiple". The default is "single".

There is also an optional **from** attribute, which is also a pattern. The exact meaning of this depends on the level.

The calculation is as follows:

level=single

1. If the current node matches the pattern, the counted node is the current node. Otherwise the counted node is the innermost ancestor of the current node that matches the pattern. If no ancestor matches the pattern, the result is zero. If the from attribute is present, the counted node must be a descendant of a node that matches the "from" pattern.
2. The result is one plus the number of elder siblings of the counted node that match the count pattern.

level=any The result is the number of nodes in the document that match the count pattern, that are at or before the current node in document order, and that follow in document order the most recent node that matches the "from" pattern, if any. Typically this is used to number, say, the diagrams or equations in a document, or in some section or chapter of a document,

	regardless of where the diagrams or equations appear in the hierarchic structure.
level=multiple	The result of this is not a single number, but a list of numbers. There is one number in the list for each ancestor of the current element that matches the count pattern and that is a descendant of the anchor element. Each number is one plus the number of elder siblings of the relevant element that match the count pattern. The order of the numbers is "outwards-in".

There is an optional **format** attribute which controls the output format. This contains an alternating sequence of format-tokens and punctuation-tokens. A format-token is any sequence of alphanumeric characters, a punctuation-token is any other sequence. The following values (among others) are supported for the format-token:

1	Sequence 1, 2, 3, ... 10, 11, 12, ...
001	Sequence 001, 002, 003, ... 010, 011, 012, ... (any number of leading zeroes)
a	Sequence a, b, c, ... aa, ab, ac, ...
A	Sequence A, B, C, ... AA, AB, AC, ...
i	Sequence i, ii, iii, iv, ... x, xi, xii, ...
I	Sequence I, II, III, IV, ... X, XI, XII, ...

There is also support for various Japanese sequences (Hiragana, Katakana, and Kanji) using the format tokens &#x3042, &#x30a2, &#x3044, &#x30a4, &#x4e00, and for Greek and Hebrew sequences.

The format token "one" gives the sequence "one", "two", "three", ... , while "ONE" gives the same in upper-case.

The default format is "1".

Actually, any sequence of ASCII digits in the format is treated in the same way: writing 999 has the same effect as writing 001. A sequence of Unicode digits other than ASCII digits (for exaple, Tibetan digits) can also be used, and will result in decimal numbering using those digits.

Similarly, any other character classified as a letter can be used, and will result in "numbering" using all consecutive Unicode letters following the one provided. For example, specifying "x" will give the sequence x, y, z, xx, xy, xz, yx, yy, yz, etc. Specifying the Greek letter alpha (²) will cause "numbering" using the Greek letters up to "Greek letter omega with tonos" (Î). Only "i" and "I" (for roman numbering), and the Japanese characters listed above, are exceptions to this rule.

Successive format-tokens in the format are used to process successive numbers in the list. If there are more format-tokens in the format than numbers in the list, the excess format-tokens and punctuation-tokens are ignored. If there are fewer format-tokens in the format than numbers in the list, the last format-token and the punctuation-token that precedes it are used to format all excess numbers, with the final punctuation-token being used only at the end.

Examples:

Number(s)	**Format**	**Result**
3	(1)	(3)
12	I	XII
2,3	1.1	2.3
2,3	1(i)	2(iii)
2,3	1.	2.3.
2,3	A.1.1	B.3.
2,3,4,5	1.1	2.3.4.5

This character may be preceded or followed by arbitrary punctuation (anything other than these characters or XML special characters such as "<") which is copied to the output verbatim. For example, the value 3 with format "(a)" produces output "(c)".

It is also possible to use xsl:number to format a number obtained from an expression. This is achieved using the value attribute of the xsl:number element. If this attribute is present, the

count, level, and from attributes are ignored.

With large numbers, the digits may be split into groups. For example, specify grouping-size="3" and grouping-separator="/" to have the number 3000000 displayed as "3/000/000".

Negative numbers are always output in conventional decimal notation, regardless of the format specified.

Example: This example outputs the title child of an H2 element preceded by a composite number formed from the sequential number of the containing H1 element and the number of the containing H2 element.

```
<xsl:template match="H2/TITLE">
    <xsl:number count="H1">.<xsl:number count="H2">
    <xsl:text> </xsl:text>
    <xsl:apply-templates/>
</xsl:template>
```

---

xsl:otherwise

The **xsl:otherwise** element is used within an **xsl:choose** element to indicate the default action to be taken if none of the other choices matches.

See xsl:choose.

---

xsl:output

The **xsl:output** element is used to control the destination and format of the principal output. It is always a top-level element immediately below the xsl:stylesheet element. There may be multiple xsl:output elements; their values are accumulated as described in the XSLT specification.

The following attributes may be specified:

method	This indicates the format or destination of the output. The value "xml" indicates XML output (though if disable-output-escaping is used there is no guarantee that it is well-formed). A value of "html" is used for HTML output. The value "text" indicates plain text output: in this case no markup may be written to the file using constructs such as literal result elements, xsl:element, xsl:attribute, or xsl:comment. The value "xx:fop" (xx is any non-default namespace) indicates that output will be directed to the Formatting Object Processor (FOP) produced by James Tauber: this must be separately installed, it is not part of SAXON. Alternatively output can be directed to a user-defined Java program by specifying the name of the class as the value of the method attribute, prefixed by a namespace prefix, for example "xx:com.me.myjava.MyEmitter". The class must be on the classpath, and must implement either the org.xml.sax.DocumentHandler interface, the org.xml.sax.ContentHandler interface, or the com.icl.saxon.output.Emitter interface. The last of these, though proprietary, is a richer interface that gives access to additional information.
indent	as in the XSLT spec: values "yes" or "no" are accepted. The indentation algorithm is different for HTML and XML. For HTML it avoids outputting extra space before or after an inline element, but will indent text as well as tags, except in elements such as PRE and SCRIPT. For XML, it avoids outputting extra whitespace except between two tags. The emphasis is on conformance rather than aesthetics!
version	Determines the version of XML or HTML to be output. Currently this is documentary only.

encoding	A character encoding, e.g. iso-8859-1 or utf-8. The value must be one recognised both by the Java run-time system and by Saxon itself: the encoding names that Saxon recognises are ASCII, US-ASCII, iso-8859-1, utf-8, utf8, KOI8R, cp1251. It is used for three distinct purposes: to control character conversion by the Java I/O routines; to determine which characters will be represented as character entities; and to document the encoding in the output file itself. The default (and fallback) is utf-8.
media-type	For example, "text/xml" or "text/html". This is largely documentary. However, the value assigned is passed back to the calling application in the OutputDetails object, where is can be accessed using the getMediaType() method. The supplied servlet application SaxonServlet uses this to set the media type in the HTTP header.
doctype-system	This is used only for XML output: it is copied into the DOCTYPE declaration as the system identifier
doctype-public	This is used only for XML output: it is copied into the DOCTYPE declaration as the public identifier. It is ignored if there is no system identifier.
omit-xml-declaration	The values are "yes" or "no". For XML output this controls whether an xml declaration should be output; the default is "no".
standalone	This is used only for XML output: if it is present, a standalone attribute is included in the XML declaration, with the value "yes" or "no".
cdata-section-elements	This is used only for XML output. It is a whitespace-separated list of element names. Character data belonging to these output elements will be written within CDATA sections.

---

xsl:param

The **xsl:param** element is used to define a formal parameter to a template, or to the stylesheet.

As a template parameter, it must be used as an immediate child of the xsl:template element. As a stylesheet parameter, it must be used as an immediate child of the xsl:stylesheet element.

There is a mandatory attribute, **name**, to define the name of the parameter. The default value of the parameter may be defined either by a select attribute, or by the contents of the xsl:param element, in the same way as for xsl:variable. The default value is ignored if an actual parameter is supplied with the same name.

---

xsl:preserve-space

The element is used at the top level of the stylesheet to define elements in the source document for which white-space nodes are significant and should be retained.

The **elements** attribute is mandatory, and defines a space-separated list of element names. The value "*" may be used to mean "all elements"; in this case any elements where whitespace is not to be preserved may be indicated by an xsl:strip-space element.

---

xsl:processing-instruction

The **xsl:processing-instruction** element can appear anywhere within an xsl:template. It causes an XML processing instruction to be output.

There is a mandatory **name** attribute which gives the name of the PI. This attribute is interpreted as an *attribute value template*, so it may contain string expressions within curly braces.

The content of the xsl:processing-instruction element is expanded to form the data part of the

PI.

For example:

```
<xsl:processing-instruction name="submit-invoice">version="1.0"</xsl:p
```

Note that special characters occurring within the PI text will *not* be escaped.

---

xsl:sort

The **xsl:sort** element is used within an **xsl:for-each** or **xsl:apply-templates** or **saxon:group** element to indicate the order in which the selected elements are processed.

The **select** attribute (default value ".") is a string expression that calculates the sort key.

The **order** attribute (values "ascending" or "descending", default "ascending") determines the sort order. There is no control over language, collating sequence, or data type.

The **data-type** attribute (values "text" or "number") determines whether collating is based on alphabetic sequence or numeric sequence.

The **case-order** attribute (values "upper-first" and "lower-first") is relevant only for data-type="text"; it determines whether uppercase letters are sorted before their lowercase equivalents, or vice-versa.

The value of the **lang** attribute can be an ISO language code such as "en" (English) or "de" (German). It determines the algorithm used for alphabetic collating. The default is based on the Java system locale. The only collating sequence supplied with the SAXON product is "en" (English), but other values may be supported by writing a user-defined comparison class. If no comparison class is found for the specified language, a default algorithm is used which simply sorts according to Unicode binary character codes. The value of lang does not have to be a recognized language code, it is also possible to use values such as "month" to select a data-type-specific collating algorithm.

Several sort keys are allowed: they are written in major-to-minor order.

Example 1: sorting with xsl:apply-templates. This example shows a template for a BOOKLIST element which processes all the child BOOK elements in order of their child AUTHOR elements; books with the same author are in descending order of the DATE attribute.

```
<xsl:template match="BOOKLIST">
    <h2>
        <xsl:apply-templates select="BOOK">
            <xsl:sort select="AUTHOR"/>
            <xsl:sort select="@DATE" order="descending" lang="Gregoria
        </xsl:apply-templates>
    </h2>
</xsl:template>
```

Example 2: sorting with xsl:for-each. This example also shows a template for a BOOKLIST element which processes all the child BOOK elements in order of their child AUTHOR elements.

```
<xsl:template match="BOOKLIST">
    <h2>
        <xsl:for-each select="BOOK">
            <xsl:sort select="AUTHOR"/>
            <p>AUTHOR: <xsl:value-of select="AUTHOR"></p>
            <p>TITLE: <xsl:value-of select="TITLE"></p>
            <hr/>
        </xsl:for-each>
    </h2>
```

```
</xsl:template>
```

xsl:script

The element is used at the top level of the stylesheet to define the implementation of extension functions. The element is defined in the draft XSLT 1.1 specification of 8 December 2000.

The **language** attribute is mandatory, and must take the value "java". The values "javascript", "ecmascript", or a QName are also permitted, but in this case Saxon ignores the xsl:script element.

The **implements-prefix** attribute is mandatory, its value must be a namespace prefix that maps to the same namespace URI as the prefix used in the extension function call.

The **src** attribute is mandatory for language="java", its value must take the form "java:fully.qualified.class.Name", for example "java:java.util.Date". It defines the class containing the implementation of extension functions that use this prefix.

The **archive** attribute is optional, its value is a space-separated list of URLs of folders or JAR files that will be searched to find the named class. If the attribute is omitted, the class is sought on the classpath.

The element name **saxon:script** may be used as a synonym for xsl:script ("saxon" being the conventional prefix for the namespace "http://icl.com/saxon"). Using the synonym enables you to define an implementation which Saxon will use, but other processors will ignore.

---

xsl:strip-space

The element is used at the top level of the stylesheet to define elements in the source document for which white-space nodes are insignificant and should be removed from the tree before processing.

The **elements** attribute is mandatory, and defines a space-separated list of element names. The value "*" may be used to mean "all elements"; in this case any elements where whitespace is not to be stripped may be indicated by an xsl:preserve-space element.

---

xsl:stylesheet

The **xsl:stylesheet** element is always the top-level element of an XSL stylesheet. The name **xsl:transform** may be used as a synonym.

The following attributes may be specified:

version	Mandatory. A value other than "1.0" invokes forwards compatibility mode.
saxon:trace	Value "yes" or "no": default no. If set to "yes", causes activation of templates to be traced on System.err for diagnostic purposes. The value may be overridden by specifying a saxon:trace attribute on the individual template.

---

xsl:template

The **xsl:template** element defines a processing rule for source elements or other nodes of a particular type.

The type of node to be processed is identified by a pattern, written in the mandatory **match** attribute. The most common form of pattern is simply an element name. However, more complex patterns may also be used: The full syntax of patterns is given in XSLT Pattern Syntax

The following examples show some of the possibilities:

Pattern	Meaning
XXX	Matches any element whose name (tag) is XXX
*	Matches any element
XXX/YYY	Matches any YYY element whose parent is a XXX
XXX//YYY	Matches any YYY element that has an ancestor named XXX
/*/XXX	Matches any XXX element that is immediatel below the root (document) element
*[@ID]	Matches any element with an ID attribute
XXX[1]	Matches any XXX element that is the first XX child of its parent element. (Note that this kin of pattern can be very inefficient: it is better t match all XXX elements with a single templat and then use xsl:if to distinguish them)
SECTION[TITLE="Contents"]	Matches any SECTION element whose first TITLE child element has the value "Contents"
A/TITLE | B/TITLE | C/TITLE	Matches any TITLE element whose parent is type A or B or C
text()	Matches any character data node
@*	Matches any attribute
/	Matches the document node

The xsl:template element has an optional **mode** attribute. If this is present, the template will only be matched when the same mode is used in the invoking **xsl:apply-templates** element.

There is also an optional **name** attribute. If this is present, the template may be invoked directly using xsl:call-template. The match attribute then becomes optional.

If there are several **xsl:template** elements that all match the same node, the one that is chosen is determined by the optional **priority** attribute: the template with highest priority wins. The priority is written as a floating-point number; the default priority is 1. If two matching templates have the same priority, the one that appears last in the stylesheet is used.

The attribute saxon:trace="yes" or "no" may be applied either at the xsl:template level or at the xsl:stylesheet level (it is treated as an inherited attribute). If the value for a template is "yes", every activation of that template results in a line of output to System.err, identifying the stylesheet template and the current node in the source document, by element type and line number. Tracing only occurs if the template is activated by matching the pattern, not if the template is called by name.

Examples:

The following examples illustrate different kinds of template and match pattern.

Example 1: a simple XSL template for a particular element. This example causes all <ptitle> elements in the source document to be output as HTML <h2> elements.

```
<xsl:template match="ptitle">
    <h2>
        <xsl:apply-templates/>
```

```
    </h2>
</xsl:template>
```

xsl:text

The **xsl:text** element causes its content to be output.

The main reasons for enclosing text within an xsl:text element is to allow white space to be output. White space nodes in the stylesheet are ignored unless they appear immediately within an xsl:text element.

The optional **disable-output-escaping** attribute may be set to "yes" or "no"; the default is "no". If set to "yes", special characters such as "<" and "&" will be output as themselves, not as entities. Be aware that in general this can produce non-well-formed XML or HTML. It is useful, however, when generating things such as ASP or JSP pages. Escaping may not be disabled when writing to a result tree fragment.

xsl:value-of

The **xsl:value-of** element evaluates an expression as a string, and outputs its value to the current output stream.

The full syntax of expressions is given in XPath Expression Syntax.

The **select** attribute identifes the expression, and is mandatory.

The optional **disable-output-escaping** attribute may be set to "yes" or "no"; the default is "no". If set to "yes", special characters such as "<" and "&" will be output as themselves, not as entities. Be aware that in general this can produce non-well-formed XML or HTML. It is useful, however, when generating things such as ASP or JSP pages. Escaping may not be disabled when writing to a result tree fragment.

If the select expression is a node-set expression that selects more than one node, only the first is considered. If it selects no node, the result is an empty string.

Here are some examples of expressions that can be used in the select attribute:

Expression	value
TITLE	The character content of the first child TITLE element if there is one
@NAME	The value of the NAME attribute of the currer element if there is one
.	The expanded character content of the curre element
../TITLE	The expanded character content of the first TITLE child of the parent element, if there is one
ancestor::SECTION/TITLE	The expanded character content of the first TITLE child of the enclosing SECTION eleme if there is one
ancestor::*/TITLE	The expanded character content of the first TITLE child of the nearest enclosing element that has a child element named TITLE
PERSON[@ID]	The content of the first child PERSON eleme having an ID attribute, if there is one

*[last()]/@ID	The value of the ID attribute of the last child element of any type, if there are any
.//TITLE	The content of the first descendant TITLE element if there is one
sum(*/@SALES)	The numeric total of the values of the SALES attributes of all child elements that have a SALES attribute

xsl:variable

The **xsl:variable** element is used to declare a variable and give it a value. If it appears at the top level (immediately within xsl:stylesheet) it declares a global variable, otherwise it declares a local variable that is visible only within the stylesheet element containing the xsl:variable declaration.

The mandatory **name** attribute defines the name of the variable.

The value of the variable may be defined either by an expression within the optional **select** attribute, or by the contents of the xsl:variable element. In the latter case the result is technically a value of type *Result Tree Fragment*, although it may be used for most practical purposes as if it were a String. (The difference is that a Result Tree Fragment may contain element start and end tags).

In standard XSL, variables once declared cannot be updated. SAXON however provides a saxon:assign extension element to circumvent this restriction. SAXON also provides an extension function to convert a result tree fragment to a node-set, allowing further processing of its contents to take place.

The value of a variable can be referenced within an expression using the syntax **$name**.

Example:

```
<xsl:variable name="title">A really exciting document"</xsl:variable>
<xsl:variable name="backcolor" expr="'#FFFFCC'" />
<xsl:template match="/*">
    <HTML><TITLE<xsl:value-of select="$title"/></TITLE>
    <BODY BGCOLOR='{$backcolor}'>
    ...

    </BODY></HTML>
</xsl:template>
```

xsl:when

The **xsl:when** element is used within an **xsl:choose** element to indicate one of a number of choices. It takes a mandatory parameter, **test**, whose value is a match pattern. If this is the first xsl:when element within the enclosing xsl:choose whose test condition matches the current element, the content of the xsl:when element is expanded, otherwise it is ignored.

xsl:with-param

The **xsl:with-param** element is used to define an actual parameter to a template. It may be used within an xsl:call-template or an xsl:apply-templates or an xsl:apply-imports element. For an example, see the **xsl:template** section.

There is a mandatory attribute, **name**, to define the name of the parameter. The value of the parameter may be defined either by a select attribute, or by the contents of the xsl:param element, in the same way as for xsl:variable.

*The parameter has no effect unless the called template includes a matching **xsl:param** element.*

Literal result elements

Any elements in the style sheet other than those listed above are assumed to be literal result elements, and are copied to the current output stream at the position in which they occur.

Attribute values within literal result elements are treated as attribute value templates: they may contain string expressions enclosed between curly braces. For the syntax of string expressions, see **xsl:value-of** above.

Where the output is HTML, certain formatting elements are recognised as empty elements: these are AREA, BASEFONT, BR, COL, FRAME, HR, IMG, INPUT, ISINDEX, LINK, META, and SYSTEM (in either upper or lower case, and optionally with attributes, of course). These should be written as empty XML elements in the stylesheet, and will be written to the HTML output stream without a closing tag.

With HTML output, if the attribute name is the same as its value, the abbreviated form of output is used: for example if <OPTION SELECTED="SELECTED"> appears in the stylesheet, it will be output as <OPTION SELECTED>.

A simple stylesheet may be created by using a literal result element as the top-level element of the stylesheet. This implicitly defines a single template with a match pattern of "/". In fact, an XHTML document constitutes a valid stylesheet which will be output as a copy of itself, regardless of the contents of the source XML document.

Michael H. Kay
4 February 2001