XSL Transformations (XSLT) Version 1.0

W3C Proposed Recommendation 8 October 1999

This version:
http://www.w3.org/TR/1999/PR-xslt-19991008
(available in XML or HTML)
Latest version:
http://www.w3.org/TR/xslt
Previous versions:
http://www.w3.org/1999/08/WD-xslt-19990813
http://www.w3.org/1999/07/WD-xslt-19990709
http://www.w3.org/TR/1999/WD-xslt-19990421
http://www.w3.org/TR/1998/WD-xsl-19981216
http://www.w3.org/TR/1998/WD-xsl-19980818
Editor:
James Clark <jjc@jclark.com>

Abstract

This specification defines the syntax and semantics of XSLT, which is a language for transforming XML documents into other XML documents.

XSLT is designed for use as part of XSL, which is a stylesheet language for XML. In addition to XSLT, XSL includes an XML vocabulary for specifying formatting. XSL specifies the styling of an XML document by using XSLT to describe how the document is transformed into another XML document that uses the formatting vocabulary.

XSLT is also designed to be used independently of XSL. However, XSLT is not intended as a completely general-purpose XML transformation language. Rather it is designed primarily for the kinds of transformations that are needed when XSLT is used as part of XSL.

Status of this document

This is a W3C Proposed Recommendation for review by W3C members and other interested parties. Please send detailed comments to xsl-editors@w3.org before 5 November 1999; archives of the comments are available. W3C members may send their formal comments, visible only to W3C staff, to w3c-xslt-review@w3.org.

This specification is a revision of the last call working draft dated 1999-08-13. The XSL Working Group anticipates no further substantial changes to this specification and encourages active implementation to test this specification during the Proposed Recommendation review period.

Publication as a Proposed Recommendation does not imply endorsement by the W3C membership. This is still a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite W3C Proposed Recommendations as other than "work in progress."

This specification has been produced as part of the Style activity.

The English version of this specification is the only normative version. However, for translations of this document, see http://www.w3.org/Style/XSL/translations.html.

Public discussion of XSL, including XSL Transformations, takes place on the XSL-List mailing list.

Table of contents

Appendices

---

1 Introduction

This specification defines the syntax and semantics of the XSLT language. A transformation in the XSLT language is expressed as a well-formed XML document [XML] conforming to the Namespaces in XML Recommendation [XML Names], which may include both elements that are defined by XSLT and elements that are not defined by XSLT. XSLT-defined elements are distinguished by belonging to a specific XML namespace (see [**2.1 XSLT Namespace**]), which is referred to in this specification as the **XSLT namespace**. Thus this specification is a definition of the syntax and semantics of the XSLT namespace.

A transformation expressed in XSLT describes rules for transforming a source tree into a result tree. The transformation is achieved by associating patterns with templates. A pattern is matched against elements in the source tree. A template is instantiated to create part of the result tree. The result tree is separate from the source tree. The structure of the result tree can be completely different from the structure of the source tree. In constructing the result tree, elements from the source tree can be filtered and reordered, and arbitrary structure can be added.

A transformation expressed in XSLT is called a stylesheet. This is because, in the case when XSLT is transforming into the XSL formatting vocabulary, the transformation functions as a stylesheet.

This document does not specify how an XSLT stylesheet is associated with an XML document. It is recommended that XSL processors support the mechanism described in [XML Stylesheet]. When this or any other mechanism yields a sequence of more than one XSLT stylesheet to be applied simultaneously to a XML document, then the effect should be the same as applying a single stylesheet that imports each member of the sequence in order (see [**2.6.2 Stylesheet Import**]).

A stylesheet contains a set of template rules. A template rule has two parts: a pattern which is matched against nodes in the source tree and a template which can be instantiated to form part of the result tree. This allows a stylesheet to be applicable to a wide class of documents that have similar source tree structures.

A template is instantiated for a particular source element to create part of the result tree. A template can contain elements that specify literal result element structure. A template can also contain elements from the XSLT namespace that are instructions for creating result tree fragments. When a template is instantiated, each instruction is executed and replaced by the result tree fragment that it creates.

Instructions can select and process descendant source elements. Processing a descendant element creates a result tree fragment by finding the applicable template rule and instantiating its template. Note that elements are only processed when they have been selected by the execution of an instruction. The result tree is constructed by finding the template rule for the root node and instantiating its template.

In the process of finding the applicable template rule, more than one template rule may have a pattern that matches a given element. However, only one template rule will be applied. The method for deciding which template rule to apply is described in [**5.5 Conflict Resolution for Template Rules**].

A single template by itself has considerable power: it can create structures of arbitrary complexity; it can pull string values out of arbitrary locations in the source tree; it can generate structures that are repeated according to the occurrence of elements in the source tree. For simple transformations where the structure of the result tree is independent of the structure of the source tree, a stylesheet can often consist of only a single template, which functions as a template for the complete result tree. Transformations on XML documents that represent data are often of this kind (see [**D.2 Data Example**]). XSLT allows a simplified syntax for such stylesheets (see [**2.3 Literal Result Element as Stylesheet**]).

When a template is instantiated, it is always instantiated with respect to a **current node** and a **current node list**. The current node is always a member of the current node list. Many operations in XSLT are relative to the current node. Only a few instructions change the current node list or the current node (see [**5 Template Rules**] and [**8 Repetition**]); during the instantiation of one of these instructions, the current node list changes to a new list of nodes and each member of this new list becomes the current node in turn; after the instantiation of the instruction is complete, the current node and current node list revert to what they were before the instruction was instantiated.

XSLT makes use of the expression language defined by [XPath] for selecting elements for processing, for conditional processing and for generating text.

XSLT provides two "hooks" for extending the language, one hook for extending the set of instruction elements used in templates and one hook for extending the set of functions used in XPath expressions. These hooks are both based on XML namespaces. This version of XSLT does not define a mechanism for implementing the hooks. See [**14 Extensions**].

> **NOTE:** The XSL WG intends to define such a mechanism in a future version of this specification or in a separate specification.

The element syntax summary notation used to describe the syntax of XSLT-defined elements is described in [**18 Notation**].

The MIME media types `text/xml` and `application/xml` [RFC2376] should be used for XSLT stylesheets. It is possible that a media type will be registered specifically for XSLT stylesheets; if and when it is, that media type may also be used.

2 Stylesheet Structure

2.1 XSLT Namespace

The XSLT namespace has the URI `http://www.w3.org/1999/XSL/Transform`.

> **NOTE:** The `1999` in the URI indicates the year in which the URI was allocated by the W3C. It does not indicate the version of XSLT being used, which is specified by attributes (see [**2.2 Stylesheet Element**] and [**2.3 Literal Result Element as Stylesheet**]).

XSLT processors must use the XML namespaces mechanism [XML Names] to recognize elements and attributes from this namespace. Elements from the XSLT namespace are recognized only in the stylesheet not in the source document. The complete list of XSLT-defined elements is specified in [**B Element Syntax Summary**]. Vendors must not extend the XSLT namespace with additional elements or attributes. Instead, any extension must be in a separate namespace. Any namespace that is used for additional instruction elements must be identified by means of the extension element mechanism specified in [**14.1 Extension Elements**].

This specification uses a prefix of `xsl:` for referring to elements in the XSLT namespace. However, XSLT stylesheets are free to use any prefix, provided that there is a namespace declaration that binds the prefix to the URI of the XSLT namespace.

An element from the XSLT namespace may have any attribute not from the XSLT namespace, provided that the expanded-name of the attribute has a non-null namespace URI. The presence of such attributes must not change the behavior of XSLT elements and functions defined in this document. Thus, an XSLT processor is always free to ignore such attributes, and must ignore such attributes without giving an error if it does not recognize the namespace URI. Such attributes can provide, for example, unique identifiers, optimization hints, or documentation.

It is an error for an element from the XSLT namespace to have attributes with expanded-names that have null namespace URIs (i.e. attributes with unprefixed names) other than attributes defined for the element in this document.

> **NOTE:** The conventions used for the names of XSLT elements, attributes and functions are that names are all lower-case, use hyphens to separate words, and use abbreviations only if they already appear in the syntax of a related language such as XML or HTML.

2.2 Stylesheet Element

```
<xsl:stylesheet
  id = id
  extension-element-prefixes = tokens
  exclude-result-prefixes = tokens
  version = number>
  <!-- Content: (xsl:import*, top-level-elements) -->
</xsl:stylesheet>

<xsl:transform
  id = id
  extension-element-prefixes = tokens
  exclude-result-prefixes = tokens
  version = number>
  <!-- Content: (xsl:import*, top-level-elements) -->
</xsl:transform>
```

A stylesheet is represented by an `xsl:stylesheet` element in an XML document. `xsl:transform` is allowed as a synonym for `xsl:stylesheet`.

An `xsl:stylesheet` element must have a `version` attribute, indicating the version of XSLT that the stylesheet requires. For this version of XSLT, the value should be `1.0`. When the value is not equal to `1.0`, forwards-compatible processing mode is enabled (see [**2.5 Forwards-Compatible Processing**]).

The `xsl:stylesheet` element may contain the following types of elements:

- `xsl:import`
- `xsl:include`
- `xsl:strip-space`
- `xsl:preserve-space`
- `xsl:output`
- `xsl:key`
- `xsl:decimal-format`
- `xsl:attribute-set`
- `xsl:variable`
- `xsl:param`
- `xsl:template`

An element occurring as a child of an `xsl:stylesheet` element is called a **top-level** element.

This example shows the structure of a stylesheet. Ellipses (`...`) indicate where attribute values or content have been omitted. Although this example shows one of each type of allowed element, stylesheets may contain zero or more of each of these elements.

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

  <xsl:include href="..."/>

  <xsl:strip-space elements="..."/>

  <xsl:preserve-space elements="..."/>

  <xsl:output method="..."/>

  <xsl:key name="..." match="..." use="..."/>
```

```
  <xsl:decimal-format name="..."/>

  <xsl:namespace-alias stylesheet-prefix="..." result-prefix="..."/>

  <xsl:attribute-set name="...">
    ...
  </xsl:attribute-set>

  <xsl:variable name="...">...</xsl:variable>

  <xsl:param name="...">...</xsl:param>

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

  <xsl:template name="...">
    ...
  </xsl:template>

</xsl:stylesheet>
```

The order in which the children of the `xsl:stylesheet` element occur is not significant except for `xsl:import` elements and for error recovery. Users are free to order the elements as they prefer, and stylesheet creation tools need not provide control over the order in which the elements occur.

In addition, the `xsl:stylesheet` element may contain any element not from the XSLT namespace, provided that the expanded-name of the element has a non-null namespace URI. The presence of such top-level elements must not change the behavior of XSLT elements and functions defined in this document; for example, it would not be permitted for such a top-level element to specify that `xsl:apply-templates` was to use different rules to resolve conflicts. Thus, an XSLT processor is always free to ignore such top-level elements, and must ignore a top-level element without giving an error if it does not recognize the namespace URI. Such elements can provide, for example,

- information used by extension elements or extension functions (see [**14 Extensions**]),
- information about what to do with the result tree,
- information about how to obtain the source tree,
- metadata about the stylesheet,
- structured documentation for the stylesheet.

2.3 Literal Result Element as Stylesheet

A simplified syntax is allowed for stylesheets that consist of only a single template for the root node. The stylesheet may consist of just a literal result element (see [**7.1.1 Literal Result Elements**]). Such a stylesheet is equivalent to a stylesheet with an `xsl:stylesheet` element containing a template rule containing the literal result element; the template rule has a match pattern of `/`. For example

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

```
      xmlns="http://www.w3.org/TR/xhtml1/strict">
  <head>
    <title>Expense Report Summary</title>
  </head>
  <body>
    <p>Total Amount: <xsl:value-of select="expense-report/total"/></p>
  </body>
</html>
```

has the same meaning as

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns="http://www.w3.org/TR/xhtml1/strict">
<xsl:template match="/">
<html>
  <head>
    <title>Expense Report Summary</title>
  </head>
  <body>
    <p>Total Amount: <xsl:value-of select="expense-report/total"/></p>
  </body>
</html>
</xsl:template>
</xsl:stylesheet>
```

A literal result element that is the document element of a stylesheet should have an `xsl:version` attribute, which indicates the version of XSLT that the stylesheet requires. For this version of XSLT, the value should be `1.0`; the value must be a Number. The `xsl:version` attribute is not required; however, it is strongly recommended. Other literal result elements may also have an `xsl:version` attribute. When the `xsl:version` attribute is not specified or is not equal to `1.0`, forwards-compatible processing mode is enabled (see [**2.5 Forwards-Compatible Processing**]).

The allowed content of a literal result element when used as a stylesheet is no different from when it occurs within a stylesheet. Thus, a literal result element used as a stylesheet cannot contain top-level elements.

> **NOTE:** If the `xsl:version` attribute is not specified on the document element, then the XSLT namespace need not be declared on the document element. Attributes on literal result elements that do not have a declaration of the XSLT namespace in scope are still treated as attribute value templates and aliased namespaces are still treated as usual (see [**7.1.1 Literal Result Elements**]).

In some situations, the only way that a system can recognize that an XML document needs to be processed by an XSLT processor as an XSLT stylesheet is by examining the XML document itself. Using the simplified syntax makes this harder, and can even make it impossible if the `xsl:version` attribute is omitted from the document element. Therefore, the simplified syntax should not be used for XSLT stylesheets that may be used in such a situation. This situation can, for example, arise when an XSLT stylesheet is transmitted as a message with a MIME media type of `text/xml` or `application/xml` to a recipient that will use the MIME media type to determine how the message is processed.

2.4 Qualified Names

The name of an internal XSLT object, specifically a named template (see [**6 Named Templates**]), a mode (see [**5.7 Modes**]), an attribute set (see [**7.1.4 Named Attribute Sets**]), a key (see [**12.2 Keys**]), a decimal-format (see [**12.3 Number Formatting**]), a variable or a parameter (see [**11 Variables and Parameters**]) is specified as a QName. If it has a prefix, then the prefix is expanded into a URI reference using the namespace declarations in effect on the attribute in which the name occurs. The expanded-name consisting of the local part of the name and the possibly null URI reference is used as the name of the object. The default namespace is *not* used for unprefixed names.

2.5 Forwards-Compatible Processing

An element enables forwards-compatible mode for itself, its attributes, its descendants and their attributes if either it is an `xsl:stylesheet` element whose `version` attribute is not equal to `1.0`, or it is a literal result element that has an `xsl:version` attribute whose value is not equal to `1.0`, or it is a literal result element that does not have an `xsl:version` attribute and that is the document element of a stylesheet using the simplified syntax (see [**2.3 Literal Result Element as Stylesheet**]). A literal result element that has an `xsl:version` attribute whose value is equal to `1.0` disables forwards-compatible mode for itself, its attributes, its descendants and their attributes.

If an element is processed in forwards-compatible mode, then:

- if it is a top-level element and XSLT 1.0 does not allow such elements as top-level elements, then the element must be ignored along with its content;

- if it is an element in a template and XSLT 1.0 does not allow such elements to occur in templates, then if the element is not instantiated, an error must not be signaled, and if the element is instantiated, the XSLT must perform fallback for the element as specified in [**15 Fallback**];

- if the element has an attribute that XSLT 1.0 does not allow the element to have or if the element has an optional attribute with a value that the XSLT 1.0 does not allow the attribute to have, then the attribute must be ignored.

Thus, any XSLT 1.0 processor must be able to process the following stylesheet without error, although the stylesheet includes elements from the XSLT namespace that are not defined in this specification:

```
<xsl:stylesheet version="1.1"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <xsl:choose>
      <xsl:when test="system-property('xsl:version') >= 1.1">
        <xsl:exciting-new-1.1-feature/>
      </xsl:when>
      <xsl:otherwise>
        <html>
        <head>
          <title>XSLT 1.1 required</title>
        </head>
        <body>
          <p>Sorry, this stylesheet requires XSLT 1.1.</p>
        </body>
        </html>
      </xsl:otherwise>
```

```
    </xsl:choose>
  </xsl:template>
</xsl:stylesheet>
```

NOTE: If a stylesheet depends crucially on a top-level element introduced by a version of XSL after 1.0, then the stylesheet can use an `xsl:message` element with `terminate="yes"` (see [**13 Messages**]) to ensure that XSLT processors implementing earlier versions of XSL will not silently ignore the top-level element. For example,

```
<xsl:stylesheet version="1.5"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:important-new-1.1-declaration/>

  <xsl:template match="/">
    <xsl:choose>
      <xsl:when test="system-property('xsl:version') &lt; 1.1">
        <xsl:message terminate="yes">
          <xsl:text>Sorry, this stylesheet requires XSLT 1.1.</xsl:text>
        </xsl:message>
      </xsl:when>
      <xsl:otherwise>
        ...
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>
  ...
</xsl:stylesheet>
```

If an expression occurs in an attribute that is processed in forwards-compatible mode, then an XSLT processor must recover from errors in the expression as follows:

- if the expression does not match the syntax allowed by the XPath grammar, then an error must not be signaled unless the expression is actually evaluated;
- if the expression calls a function with an unprefixed name that is not part of the XSLT library, then an error must not be signaled unless the function is actually called;
- if the expression calls a function with a number of arguments that XSLT does not allow or with arguments of types that XSLT does not allow, then an error must not be signaled unless the function is actually called.

2.6 Combining Stylesheets

XSLT provides two mechanisms to combine stylesheets:

- an inclusion mechanism that allows stylesheets to be combined without changing the semantics of the stylesheets being combined, and
- an import mechanism that allows stylesheets to override each other.

2.6.1 Stylesheet Inclusion

```
<!-- Category: top-level-element -->
```

```
<xsl:include
  href = uri-reference />
```

An XSLT stylesheet may include another XSLT stylesheet using an `xsl:include` element. The `xsl:include` element has an `href` attribute whose value is a URI reference identifying the stylesheet to be included. A relative URI is resolved relative to the base URI of the `xsl:include` element (see [**3.2 Base URI**]).

The `xsl:include` element is only allowed as a top-level element.

The inclusion works at the XML tree level. The resource located by the `href` attribute value is parsed as an XML document, and the children of the `xsl:stylesheet` element in this document replace the `xsl:include` element in the including document. The fact that template rules or definitions are included does not affect the way they are processed.

The included stylesheet may use the simplified syntax described in [**2.3 Literal Result Element as Stylesheet**]. The included stylesheet is treated the same as the equivalent `xsl:stylesheet` element.

It is an error if a stylesheet directly or indirectly includes itself.

> **NOTE:** Including a stylesheet multiple times can cause errors because of duplicate definitions. Such multiple inclusions are less obvious when they are indirect. For example, if stylesheet *B* includes stylesheet *A*, stylesheet *C* includes stylesheet *A*, and stylesheet *D* includes both stylesheet *B* and stylesheet *C*, then *A* will be included indirectly by *D* twice. If all of *B*, *C* and *D* are used as independent stylesheets, then the error can be avoided by separating everything in *B* other than the inclusion of *A* into a separate stylesheet *B'* and changing *B* to contain just inclusions of *B'* and *A*, similarly for *C*, and then changing *D* to include *A*, *B'*, *C'*.

2.6.2 Stylesheet Import

```
<xsl:import
  href = uri-reference />
```

An XSLT stylesheet may import another XSLT stylesheet using an `xsl:import` element. Importing a stylesheet is the same as including it (see [**2.6.1 Stylesheet Inclusion**]) except that definitions and template rules in the importing stylesheet take precedence over template rules and definitions in the imported stylesheet; this is described in more detail below. The `xsl:import` element has an `href` attribute whose value is a URI reference identifying the stylesheet to be imported. A relative URI is resolved relative to the base URI of the `xsl:import` element (see [**3.2 Base URI**]).

The `xsl:import` element is only allowed as a top-level element. The `xsl:import` element children must precede all other element children of an `xsl:stylesheet` element, including any `xsl:include` element children. When `xsl:include` is used to include a stylesheet, any `xsl:import` elements in the included document are moved up in the including document to after any existing `xsl:import` elements in the including document.

For example,

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:import href="article.xsl"/>
  <xsl:import href="bigfont.xsl"/>
  <xsl:attribute-set name="note-style">
    <xsl:attribute name="font-style">italic</xsl:attribute>
  </xsl:attribute-set>
</xsl:stylesheet>
```

The `xsl:stylesheet` elements encountered during processing of a stylesheet that contains `xsl:import` elements are treated as forming an **import tree**. In the import tree, each `xsl:stylesheet` element has one import child for each `xsl:import` element that it contains. Any `xsl:include` elements are resolved before constructing the import tree. An `xsl:stylesheet` element in the import tree is defined to have lower **import precedence** than another `xsl:stylesheet` element in the import tree if it would be visited before that `xsl:stylesheet` element in a post-order traversal of the import tree (i.e. a traversal of the import tree in which an `xsl:stylesheet` element is visited after its import children). Each definition and template rule has import precedence determined by the `xsl:stylesheet` element that contains it.

For example, suppose

- stylesheet *A* imports stylesheets *B* and *C* in that order;
- stylesheet *B* imports stylesheet *D*;
- stylesheet *C* imports stylesheet *E*.

Then the order of import precedence (lowest first) is *D*, *B*, *E*, *C*, *A*.

> **NOTE:** Since `xsl:import` elements are required to occur before any definitions or template rules, an implementation that processes imported stylesheets at the point at which it encounters the `xsl:import` element will encounter definitions and template rules in increasing order of import precedence.

In general, a definition or template rule with higher import precedence takes precedence over a definition or template rule with lower import precedence. This is defined in detail for each kind of definition and for template rules.

It is an error if a stylesheet directly or indirectly imports itself. Apart from this, the case where a stylesheet with a particular URI is imported in multiple places is not treated specially. The import tree will have a separate `xsl:stylesheet` for each place that it is imported.

> **NOTE:** If `xsl:apply-imports` is used (see [**5.6 Overriding Template Rules**]), the behavior may be different from the behavior if the stylesheet had been imported only at the place with the highest import precedence.

2.7 Embedding Stylesheets

Normally an XSLT stylesheet is a complete XML document with the `xsl:stylesheet` element as the document element. However, an XSLT stylesheet may also be embedded in another resource. Two forms of embedding are possible:

- the XSLT stylesheet may be textually embedded in a non-XML resource, or
- the `xsl:stylesheet` element may occur in an XML document other than as the document element.

To facilitate the second form of embedding, the `xsl:stylesheet` element is allowed to have an ID attribute that specifies a unique identifier.

> **NOTE:** In order for such an attribute to be used with the XPath **id** function, it must actually be declared in the DTD as being an ID.

The following example shows how the `xml-stylesheet` processing instruction [XML Stylesheet] can be used to allow a document to contain its own stylesheet. The URI reference uses a relative URI with a fragment identifier to locate the `xsl:stylesheet` element:

```
<?xml-stylesheet type="text/xml" href="#style1"?>
<!DOCTYPE doc SYSTEM "doc.dtd">
<doc>
<head>
<xsl:stylesheet id="style1"
                version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:fo="http://www.w3.org/1999/XSL/Format">
<xsl:import href="doc.xsl"/>
<xsl:template match="id('foo')">
  <fo:block font-weight="bold"><xsl:apply-templates/></fo:block>
</xsl:template>
<xsl:template match="xsl:stylesheet">
  <!-- ignore -->
</xsl:template>
</xsl:stylesheet>
</head>
<body>
<para id="foo">
...
</para>
</body>
</doc>
```

> **NOTE:** A stylesheet that is embedded in the document to which it is to be applied or that may be included or imported into an stylesheet that is so embedded typically needs to contain a template rule that specifies that `xsl:stylesheet` elements are to be ignored.

3 Data Model

The data model used by XSLT is the same as that used by XPath with the additions described in this section. XSLT operates on source, result and stylesheet documents using the same data model. Any two XML documents that have the same tree will be treated the same by XSLT.

Processing instructions and comments in the stylesheet are ignored: the stylesheet is treated as if neither processing instruction nodes nor comment nodes were included in the tree that represents the stylesheet.

3.1 Root Node Children

The normal restrictions on the children of the root node are relaxed for the result tree. The result tree may have any sequence of nodes as children that would be possible for an element node. In particular, it may have text node children, and any number of element node children. When written out using the XML output method (see [**16 Output**]), it is possible that a result tree will not be a well-formed XML document; however, it will always be a well-formed external general parsed entity.

When the source tree is created by parsing a well-formed XML document, the root node of the source tree will automatically satisfy the normal restrictions of having no text node children and exactly one element child. When the source tree is created in some other way, for example by using the DOM, the usual restrictions are relaxed for the source tree as for the result tree.

3.2 Base URI

Every node also has an associated URI called its base URI, which is used for resolving attribute values that represent relative URIs into absolute URIs. If an element or processing instruction occurs in an external entity, the base URI of that element or processing instruction is the URI of the external entity; otherwise, the base URI is the base URI of the document. The base URI of the document node is the URI of the document entity. The base URI for a text node, a comment node, an attribute node or a namespace node is the base URI of the parent of the node.

3.3 Unparsed Entities

The root node has a mapping that gives the URI for each unparsed entity declared in the document’s DTD. The URI is generated from the system identifier and public identifier specified in the entity declaration. The XSLT processor may use the public identifier to generate a URI for the entity instead of the URI specified in the system identifier. If the XSLT processor does not use the public identifier to generate the URI, it must use the system identifier; if the system identifier is a relative URI, it must be resolved into an absolute URI using the URI of the resource containing the entity declaration as the base URI [RFC2396].

3.4 Whitespace Stripping

After the tree for a source document or stylesheet document has been constructed, but before it is otherwise processed by XSLT, some text nodes are stripped. A text node is never stripped unless it contains only whitespace characters. Stripping the text node removes the text node from the tree. The stripping process takes as input a set of element names for which whitespace must be preserved. The stripping process is applied to both stylesheets and source documents, but the set of whitespace-preserving element names is determined differently for stylesheets and for source documents.

A text node is preserved if any of the following apply:

- The element name of the parent of the text node is in the set of whitespace-preserving element names.

- The text node contains at least one non-whitespace character. As in XML, a whitespace character is #x20, #x9, #xD or #xA.
- An ancestor element of the text node has an `xml:space` attribute with a value of `preserve`, and no closer ancestor element has `xml:space` with a value of `default`.

Otherwise, the text node is stripped.

The `xml:space` attributes are not stripped from the tree.

> **NOTE:** This implies that if an `xml:space` attribute is specified on a literal result element, it will be included in the result.

For stylesheets, the set of whitespace-preserving element names consists of just `xsl:text`.

```
<!-- Category: top-level-element -->
<xsl:strip-space
  elements = tokens />

<!-- Category: top-level-element -->
<xsl:preserve-space
  elements = tokens />
```

For source documents, the set of whitespace-preserving element names is specified by `xsl:strip-space` and `xsl:preserve-space` top-level elements. These elements each have an `elements` attribute whose value is a whitespace-separated list of NameTests. Initially, the set of whitespace-preserving element names contains all element names. If an element name matches a NameTest in an `xsl:strip-space` element, then it is removed from the set of whitespace-preserving element names. If an element name matches a NameTest in an `xsl:preserve-space` element, then it is added to the set of whitespace-preserving element names. An element matches a NameTest if and only if the NameTest would be true for the element as an XPath node test. Conflicts between matches to `xsl:strip-space` and `xsl:preserve-space` elements are resolved the same way as conflicts between template rules (see [**5.5 Conflict Resolution for Template Rules**]). Thus, the applicable match for a particular element name is determined as follows:

- First, any match with lower import precedence than another match is ignored.
- Next, any match with a NameTest that has a lower default priority than the default priority of the NameTest of another match is ignored.

It is an error if this leaves more than one match. An XSLT processor may signal the error; if it does not signal the error, it must recover by choosing, from amongst the matches that are left, the one that occurs last in the stylesheet.

4 Expressions

XSLT uses the expression language defined by XPath [XPath]. Expressions are used in XSLT for a variety of purposes including:

- selecting nodes for processing;
- specifying conditions for different ways of processing a node;
- generating text to be inserted in the result tree.

An **expression** must match the XPath production Expr.

Expressions occur as the value of certain attributes on XSLT-defined elements and within curly braces in attribute value templates.

In XSLT, an outermost expression (i.e. an expression that is not part of another expression) gets its context as follows:

- the context node comes from the current node

- the context position comes from the position of the current node in the current node list; the first position is 1

- the context size comes from the size of the current node list

- the variable bindings are the bindings in scope on the element which has the attribute in which the expression occurs (see [**11 Variables and Parameters**])

- the set of namespace declarations are those in scope on the element which has the attribute in which the expression occurs; the default namespace (as declared by `xmlns`) is not part of this set

- the function library consists of the core function library together with the additional functions defined in [**12 Additional Functions**] and extension functions as described in [**14 Extensions**]; it is an error for an expression to include a call to any other function

5 Template Rules

5.1 Processing Model

A list of source nodes is processed to create a result tree fragment. The result tree is constructed by processing a list containing just the root node. A list of source nodes is processed by appending the result tree structure created by processing each of the members of the list in order. A node is processed by finding all the template rules with patterns that match the node, and choosing the best amongst them; the chosen rule's template is then instantiated with the node as the current node and with the list of source nodes as the current node list. A template typically contains instructions that select an additional list of source nodes for processing. The process of matching, instantiation and selection is continued recursively until no new source nodes are selected for processing.

Implementations are free to process the source document in any way that produces the same result as if it were processed using this processing model.

5.2 Patterns

Template rules identify the nodes to which they apply by using a **pattern**. As well as being used in template rules, patterns are used for numbering (see [**7.7 Numbering**]) and for declaring keys (see [**12.2 Keys**]). A pattern specifies a set of conditions on a node. A node that satisfies the conditions matches the pattern; a node that does not satisfy the conditions does not match the pattern. The syntax for patterns is a subset of the syntax for expressions. In particular, location paths that meet certain restrictions can be used as patterns. An expression that is also a pattern always evaluates to an object of type node-set. A node matches a pattern if the node is a member of the result of evaluating the pattern as an expression with respect to some possible context; the possible contexts are those whose context node is the node being matched or one of its ancestors.

Here are some examples of patterns:

- `para` matches any `para` element
- `*` matches any element
- `chapter|appendix` matches any `chapter` element and any `appendix` element
- `olist/item` matches any `item` element with an `olist` parent
- `appendix//para` matches any `para` element with an `appendix` ancestor element
- `/` matches the root node
- `text()` matches any text node
- `processing-instruction()` matches any processing instruction
- `node()` matches any node other than an attribute node and the root node
- `id("W11")` matches the element with unique ID `W11`
- `para[1]` matches any `para` element that is the first `para` child element of its parent
- `*[position()=1 and self::para]` matches any `para` element that is the first child element of its parent
- `para[last()=1]` matches any `para` element that is the only `para` child element of its parent
- `items/item[position()>1]` matches any `item` element that has a `items` parent and that is not the first `item` child of its parent
- `item[position() mod 2 = 1]` would be true for any `item` element that is an odd-numbered `item` child of its parent.
- `div[@class="appendix"]//p` matches any `p` element with a `div` ancestor element that has a `class` attribute with value `appendix`

- `@class` matches any `class` attribute (*not* any element that has a `class` attribute)
- `@*` matches any attribute

A pattern must match the grammar for Pattern. A Pattern is a set of location path patterns separated by `|`. A location path pattern is a location path whose steps all use only the `child` or `attribute` axes. Although patterns must not use the `descendant-or-self` axis, patterns may use the `//` operator as well as the `/` operator. Location path patterns can also start with an **id** or **key** function call with a literal argument. Predicates in a pattern can use arbitrary expressions just like predicates in a location path.

Patterns

[1]	Pattern	::=	LocationPathPattern
			| Pattern '|' LocationPathPattern
[2]	LocationPathPattern	::=	'/' RelativePathPattern?
			| IdKeyPattern (('/' | '//') RelativePathPattern)?
			| '//'? RelativePathPattern
[3]	IdKeyPattern	::=	'id' '(' Literal ')'
			| 'key' '(' Literal ',' Literal ')'
[4]	RelativePathPattern	::=	StepPattern
			| RelativePathPattern '/' StepPattern
			| RelativePathPattern '//' StepPattern
[5]	StepPattern	::=	ChildOrAttributeAxisSpecifier NodeTest Predicate*
[6]	ChildOrAttributeAxisSpecifier	::=	AbbreviatedAxisSpecifier
			| ('child' | 'attribute') '::'

A pattern is defined to match a node if and only if there is possible context such that when the pattern is evaluated as an expression with that context, the node is a member of the resulting node-set. When a node is being matched, the possible contexts have a context node that is the node being matched or any ancestor of that node, and a context node list containing just the context node.

For example, `p` matches any `p` element, because for any `p` if the expression `p` is evaluated with the parent of the `p` element as context the resulting node-set will contain that `p` element as one of its members.

> **NOTE:** This matches even a `p` element that is the document element, since the document root is the parent of the document element.

Although the semantics of patterns are specified indirectly in terms of expression evaluation, it is easy to understand the meaning of a pattern directly without thinking in terms of expression evaluation. In a pattern, `|` indicates alternatives; a pattern with one or more `|` separated alternatives matches if any one of the alternative matches. A pattern that consists of a sequence of StepPatterns separated by `/` or `//` is matched from right to left. The pattern only matches if the rightmost StepPattern matches and a suitable element matches the rest of the pattern; if the separator is `/` then only the parent is a suitable element; if the separator is `//`, then any ancestor is a suitable element. A StepPattern that uses the child axis matches if the NodeTest is true for the node and the node is not an attribute node. A StepPattern that uses the attribute axis matches if the NodeTest is true for the node and the node is an attribute node. When `[]` is present, then the first PredicateExpr in a StepPattern is evaluated with the node being matched as the context node and the siblings of the context node that match the NodeTest as the context node list, unless the node being matched is an attribute node, in which case the context node list is all the attributes that have the same parent as the attribute being matched and that match the NameTest.

For example

```
appendix//ulist/item[position()=1]
```

matches a node if and only if all of the following are true:

- the NodeTest `item` is true for the node and the node is not an attribute; in other words the node is an `item` element
- evaluating the PredicateExpr `position()=1` with the node as context node and the siblings of the node that are `item` elements as the context node list yields true
- the node has a parent that matches `appendix//ulist`; this will be true if the parent is a `ulist` element that has an `appendix` ancestor element.

5.3 Defining Template Rules

```
<!-- Category: top-level-element -->
<xsl:template
  match = pattern
  name = qname
  priority = number
  mode = qname>
  <!-- Content: (xsl:param*, template) -->
</xsl:template>
```

A template rule is specified with the `xsl:template` element. The `match` attribute is a Pattern that identifies the source node or nodes to which the rule applies. The `match` attribute is required unless the `xsl:template` element has a `name` attribute (see [**6 Named Templates**]). It is an error for the value of the `match` attribute to contain a VariableReference. The content of the `xsl:template` element is the template that is instantiated when the template rule is applied.

For example, an XML document might contain:

```
This is an <emph>important</emph> point.
```

The following template rule matches `emph` elements and produces a `fo:inline-sequence` formatting object with a `font-weight` property of `bold`.

```
<xsl:template match="emph">
  <fo:inline-sequence font-weight="bold">
    <xsl:apply-templates/>
  </fo:inline-sequence>
</xsl:template>
```

NOTE: Examples in this document use the `fo:` prefix for the namespace `http://www.w3.org/1999/XSL/Format`, which is the namespace of the formatting objects defined in [XSL].

As described next, the `xsl:apply-templates` element recursively processes the children of the source element.

5.4 Applying Template Rules

```
<!-- Category: instruction -->
<xsl:apply-templates
  select = node-set-expression
  mode = qname>
  <!-- Content: (xsl:sort | xsl:with-param)* -->
</xsl:apply-templates>
```

This example creates a block for a `chapter` element and then processes its immediate children.

```
<xsl:template match="chapter">
  <fo:block>
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>
```

In the absence of a `select` attribute, the `xsl:apply-templates` instruction processes all of the children of the current node, including text nodes. However, text nodes that have been stripped as specified in [**3.4 Whitespace Stripping**] will not be processed. If stripping of whitespace nodes has not been enabled for an element, then all whitespace in the content of the element will be processed as text, and thus whitespace between child elements will count in determining the position of a child element as returned by the **position** function.

A `select` attribute can be used to process nodes selected by an expression instead of processing all children. The value of the `select` attribute is an expression. The expression must evaluate to a node-set. The selected set of nodes is processed in document order, unless a sorting specification is present (see [**10 Sorting**]). The following example processes all of the `author` children of the `author-group`:

```
<xsl:template match="author-group">
  <fo:inline-sequence>
    <xsl:apply-templates select="author"/>
  </fo:inline-sequence>
</xsl:template>
```

The following example processes all of the `given-name`s of the `author`s that are children of `author-group`:

```
<xsl:template match="author-group">
  <fo:inline-sequence>
    <xsl:apply-templates select="author/given-name"/>
  </fo:inline-sequence>
</xsl:template>
```

This example processes all of the `heading` descendant elements of the `book` element.

```
<xsl:template match="book">
  <fo:block>
    <xsl:apply-templates select=".//heading"/>
  </fo:block>
</xsl:template>
```

It is also possible to process elements that are not descendants of the current node. This example assumes that a `department` element has `group` children and `employee` descendants. It finds an employee's department and then processes the `group` children of the `department`.

```
<xsl:template match="employee">
  <fo:block>
    Employee <xsl:apply-templates select="name"/> belongs to group
    <xsl:apply-templates select="ancestor::department/group"/>
  </fo:block>
</xsl:template>
```

Multiple `xsl:apply-templates` elements can be used within a single template to do simple reordering. The following example creates two HTML tables. The first table is filled with domestic sales while the second table is filled with foreign sales.

```
<xsl:template match="product">
  <table>
    <xsl:apply-templates select="sales/domestic"/>
  </table>
  <table>
    <xsl:apply-templates select="sales/foreign"/>
  </table>
</xsl:template>
```

> **NOTE:** It is possible for there to be two matching descendants where one is a descendant of the other. This case is not treated specially: both descendants will be processed as usual. For example, given a source document
>
> ```
> <doc> </doc>
> ```
>
> the rule
>
> ```
> <xsl:template match="doc">
>   <xsl:apply-templates select=".//div"/>
> </xsl:template>
> ```
>
> will process both the outer `div` and inner `div` elements.

> **NOTE:** Typically, `xsl:apply-templates` is used to process only nodes that are descendants of the current node. Such use of `xsl:apply-templates` cannot result in non-terminating processing loops. However, when `xsl:apply-templates` is used to process elements that are not descendants of the current node, the possibility arises of non-terminating loops. For example,
>
> ```
> <xsl:template match="foo">
>   <xsl:apply-templates select="."/>
> </xsl:template>
> ```
>
> Implementations may be able to detect such loops in some cases, but the possibility exists

that a stylesheet may enter a non-terminating loop that an implementation is unable to detect. This may present a denial of service security risk.

5.5 Conflict Resolution for Template Rules

It is possible for a source node to match more than one template rule. The template rule to be used is determined as follows:

1. First, all matching template rules that have lower import precedence than the matching template rule or rules with the highest import precedence are eliminated from consideration.

2. Next, all matching template rules that have lower priority than the matching template rule or rules with the highest priority are eliminated from consideration. The priority of a template rule is specified by the `priority` attribute on the template rule. The value of this must be a real number (positive or negative), matching the production Number with an optional leading minus sign (`-`). The **default priority** is computed as follows:

   - If the pattern contains multiple alternatives separated by `|`, then it is treated equivalently to a set of template rules, one for each alternative.

   - If the pattern has the form of a QName preceded by a ChildOrAttributeAxisSpecifier or has the form `processing-instruction(`Literal`)` preceded by a ChildOrAttributeAxisSpecifier, then the priority is 0.

   - If the pattern has the form NCName`:*` preceded by a ChildOrAttributeAxisSpecifier, then the priority is -0.25.

   - Otherwise, if the pattern consists of just a NodeTest preceded by a ChildOrAttributeAxisSpecifier, then the priority is -0.5.

   - Otherwise, the priority is 0.5.

   Thus, the most common kind of pattern (a pattern that tests for a node with a particular type and a particular expanded-name) has priority 0. The next less specific kind of pattern (a pattern that tests for a node with a particular type and an expanded-name with a particular namespace URI) has priority -0.25. Patterns less specific than this (patterns that just tests for nodes with particular types) have priority -0.5. Patterns more specific than the most common kind of pattern have priority 0.5.

It is an error if this leaves more than one matching template rule. An XSLT processor may signal the error; if it does not signal the error, it must recover by choosing, from amongst the matching template rules that are left, the one that occurs last in the stylesheet.

5.6 Overriding Template Rules

```
<!-- Category: instruction -->
<xsl:apply-imports />
```

A template rule that is being used to override a template rule in an imported stylesheet (see [**5.5 Conflict Resolution for Template Rules**]) can use the `xsl:apply-imports` element to invoke the overridden template rule.

At any point in the processing of a stylesheet, there is a **current template rule**. Whenever a template rule is chosen by matching a pattern, the template rule becomes the current template rule for the instantiation of the rule's template. When an `xsl:for-each` element is instantiated, the current template rule becomes null for the instantiation of the content of the `xsl:for-each` element.

`xsl:apply-imports` processes the current node using only template rules that were imported into the stylesheet element containing the current template rule; the node is processed in the current template rule's mode. It is an error if `xsl:apply-imports` is instantiated when the current template rule is null.

For example, suppose the stylesheet `doc.xsl` contains a template rule for `example` elements:

```
<xsl:template match="example">
  <pre><xsl:apply-templates/></pre>
</xsl:template>
```

Another stylesheet could import `doc.xsl` and modify the treatment of `example` elements as follows:

```
<xsl:import href="doc.xsl"/>

<xsl:template match="example">
  <div style="border: solid red">
     <xsl:apply-imports/>
  </div>
</xsl:template>
```

The combined effect would be to transform an `example` into an element of the form:

```
<div style="border: solid red"><pre>...</pre></div>
```

5.7 Modes

Modes allow an element to be processed multiple times, each time producing a different result.

Both `xsl:template` and `xsl:apply-templates` have an optional `mode` attribute. The value of the `mode` attribute is a QName, which is expanded as described in [**2.4 Qualified Names**]. If `xsl:template` does not have a `match` attribute, it must not have a `mode` attribute. If an `xsl:apply-templates` element has a `mode` attribute, then it applies only to those template rules from `xsl:template` elements that have a `mode` attribute with the same value; if an `xsl:apply-templates` element does not have a `mode` attribute, then it applies only to those template rules from `xsl:template` elements that do not have a `mode` attribute.

5.8 Built-in Template Rules

There is a built-in template rule to allow recursive processing to continue in the absence of a successful pattern match by an explicit template rule in the stylesheet. This template rule applies to both element nodes and the root node. The following shows the equivalent of the built-in template rule:

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

There is also a built-in template rule for each mode, which allows recursive processing to continue in the same mode in the absence of a successful pattern match by an explicit template rule in the stylesheet. This template rule applies to both element nodes and the root node. The following shows the equivalent of the built-in template rule for mode *m*.

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

There is also a built-in template rule for text and attribute nodes that copies text through:

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

The built-in template rule for processing instructions and comments is to do nothing.

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

The built-in template rule for namespace nodes is also to do nothing. There is no pattern that can match a namespace node; so, the built-in template rule is the only template rule that is applied for namespace nodes.

The built-in template rules are treated as if they were imported implicitly before the stylesheet and so have lower import precedence than all other template rules. Thus, the author can override a built-in template rule by including an explicit template rule.

6 Named Templates

```
<!-- Category: instruction -->
<xsl:call-template
  name = qname>
  <!-- Content: xsl:with-param* -->
</xsl:call-template>
```

Templates can be invoked by name. An `xsl:template` element with a `name` attribute specifies a named template. The value of the `name` attribute is a QName, which is expanded as described in [**2.4 Qualified Names**]. If an `xsl:template` element has a `name` attribute, it may, but need not, also have a `match` attribute. An `xsl:call-template` element invokes a template by name; it has a required `name` attribute that identifies the template to be invoked. Unlike `xsl:apply-templates`, `xsl:call-template` does not change the current node or the current node list.

The `match` and `mode` attributes on an `xsl:template` element do not affect whether the template is invoked by an `xsl:call-template` element. Similarly, the `name` attribute on an `xsl:template` element does not affect whether the template is invoked by an `xsl:apply-templates` element.

It is an error if a stylesheet contains more than one template with the same name and same import precedence.

7 Creating the Result Tree

This section describes instructions that directly create nodes in the result tree.

7.1 Creating Elements and Attributes

7.1.1 Literal Result Elements

In a template, an element in the stylesheet that does not belong to the XSLT namespace and that is not an extension element (see [**14.1 Extension Elements**]) is instantiated to create an element node with the same expanded-name. The content of the element is a template, which is instantiated to give the content of the created element node. The created element node will have the attribute nodes that were present on the element node in the stylesheet tree, other than attributes with names in the XSLT namespace.

The created element node will also have a copy of the namespace nodes that were present on the element node in the stylesheet tree with the exception of any namespace node whose string-value is the XSLT namespace URI (`http://www.w3.org/1999/XSL/Transform`), a namespace URI declared as an extension namespace (see [**14.1 Extension Elements**]), or a namespace URI designated as an excluded namespace. A namespace URI is designated as an excluded namespace by using an `exclude-result-prefixes` attribute on an `xsl:stylesheet` element or an `xsl:exclude-result-prefixes` attribute on a literal result element. The value of both these attributes is a whitespace-separated list of namespace prefixes. The namespace bound to each of the prefixes is designated as an excluded namespace. It is an error if there is no namespace bound to the prefix on the element bearing the `exclude-result-prefixes` or `xsl:exclude-result-prefixes` attribute. The default namespace (as declared by `xmlns`) may be designated as an excluded namespace by including `#default` in the list of namespace prefixes. The designation of a namespace as an excluded namespace is effective within the subtree of the stylesheet rooted at the element bearing the `exclude-result-prefixes` or `xsl:exclude-result-prefixes` attribute.

> **NOTE:** When a stylesheet uses a namespace declaration only for the purposes of addressing the source tree, specifying the prefix in the `exclude-result-prefixes` attribute will avoid superfluous namespace declarations in the result tree.

The value of an attribute of a literal result element is interpreted as an attribute value template: it can contain expressions contained in curly braces (`{}`).

A namespace URI in the stylesheet tree that is being used to specify a namespace URI in the result tree is called a **literal namespace URI**. This applies to:

- the namespace URI in the expanded-name of a literal result element in the stylesheet
- the namespace URI in the expanded-name of an attribute specified on a literal result element in the stylesheet
- the string-value of a namespace node on a literal result element in the stylesheet

```
<!-- Category: top-level-element -->
<xsl:namespace-alias
  stylesheet-prefix = prefix | "#default"
  result-prefix = prefix | "#default" />
```

A stylesheet can use the `xsl:namespace-alias` element to declare that one namespace URI is an **alias** for another namespace URI. When a literal namespace URI has been declared to be an alias for another namespace URI, then the namespace URI in the result tree will be the namespace URI that the literal namespace URI is an alias for, instead of the literal namespace URI itself. The `xsl:namespace-alias` element declares that the namespace URI bound to the prefix specified by the `stylesheet-namespace` attribute is an alias for the namespace URI bound to the prefix specified by the `result-namespace` attribute. Thus, the `stylesheet-prefix` attribute specifies the namespace URI that will appear in the stylesheet, and the `result-prefix` attribute specifies the corresponding namespace URI that will appear in the result tree. The default namespace (as declared by `xmlns`) may be specified by using `#default` instead of a prefix. If a namespace URI is declared to be an alias for multiple different namespace URIs, then the declaration with the highest import precedence is used. It is an error if there is more than one such declaration. An XSLT processor may signal the error; if it does not signal the error, it must recover by choosing, from amongst the declarations with the highest import precedence, the one that occurs last in the stylesheet.

When literal result elements are being used to create element, attribute, or namespace nodes that use the XSLT namespace URI, the stylesheet must use an alias. For example, the stylesheet

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

<xsl:namespace-alias stylesheet-prefix="axsl" result-prefix="xsl"/>

<xsl:template match="/">
  <axsl:stylesheet>
    <xsl:apply-templates/>
  </axsl:stylesheet>
</xsl:template>

<xsl:template match="block">
  <axsl:template match="{.}">
     <fo:block><axsl:apply-templates/></fo:block>
  </axsl:template>
</xsl:template>

</xsl:stylesheet>
```

will generate an XSLT stylesheet from a document of the form:

```
<elements>
<block>p</block>
<block>h1</block>
<block>h2</block>
<block>h3</block>
<block>h4</block>
</elements>
```

NOTE: It may be necessary also to use aliases for namespaces other than the XSLT namespace URI. For example, literal result elements belonging to a namespace dealing with digital signatures might cause XSLT stylesheets to be mishandled by general-purpose security software; using an alias for the namespace would avoid the possibility of such mishandling.

7.1.2 Creating Elements with `xsl:element`

```
<!-- Category: instruction -->
<xsl:element
  name = { qname }
  namespace = { uri-reference }
  use-attribute-sets = qnames>
  <!-- Content: template -->
</xsl:element>
```

The `xsl:element` element allows an element to be created with a computed name. The expanded-name of the element to be created is specified by a required `name` attribute and an optional `namespace` attribute. The content of the `xsl:element` element is a template for the attributes and children of the created element.

The `name` attribute is interpreted as an attribute value template. It is an error if the string that results from instantiating the attribute value template is not a QName. An XSLT processor may signal the error; if it does not signal the error, then it must recover by making the the result of instantiating the `xsl:element` element be the sequence of nodes created by instantiating the content of the `xsl:element` element, excluding any initial attribute nodes. If the `namespace` attribute is not present then the QName is expanded into an expanded-name using the namespace declarations in effect for the `xsl:element` element, including any default namespace declaration.

If the `namespace` attribute is present, then it also is interpreted as an attribute value template. The string that results from instantiating the attribute value template should be a URI reference. It is not an error if the string is not a syntactically legal URI reference. If the string is empty, then the expanded-name of the element has a null namespace URI. Otherwise, the string is used as the namespace URI of the expanded-name of the element to be created. The local part of the QName specified by the `name` attribute is used as the local part of the expanded-name of the element to be created.

XSLT processors may make use of the prefix of the QName specified in the `name` attribute when selecting the prefix used for outputting the created element as XML; however, they are not required to do so.

7.1.3 Creating Attributes with `xsl:attribute`

```
<!-- Category: instruction -->
<xsl:attribute
  name = { qname }
  namespace = { uri-reference }>
  <!-- Content: template -->
</xsl:attribute>
```

The `xsl:attribute` element can be used to add attributes to result elements whether created by literal result elements in the stylesheet or by instructions such as `xsl:element`. The expanded-name of the attribute to be created is specified by a required `name` attribute and an optional `namespace` attribute. Instantiating an `xsl:attribute` element adds an attribute node to the containing result element node. The content of the `xsl:attribute` element is a template for the value of the created attribute.

The `name` attribute is interpreted as an attribute value template. It is an error if the string that results from instantiating the attribute value template is not a QName or is the string `xmlns`. An XSLT processor may signal the error; if it does not signal the error, it must recover by not adding the attribute to the result tree. If the `namespace` attribute is not present, then the QName is expanded into an expanded-name using the namespace declarations in effect for the `xsl:attribute` element, *not* including any default namespace declaration.

If the `namespace` attribute is present, then it also is interpreted as an attribute value template. The string that results from instantiating it should be a URI reference. It is not an error if the string is not a syntactically legal URI reference. If the string is empty, then the expanded-name of the attribute has a null namespace URI. Otherwise, the string is used as the namespace URI of the expanded-name of the attribute to be created. The local part of the QName specified by the `name` attribute is used as the local part of the expanded-name of the attribute to be created.

XSLT processors may make use of the prefix of the QName specified in the `name` attribute when selecting the prefix used for outputting the created attribute as XML; however, they are not required to do so and, if the prefix is `xmlns`, they must not do so. Thus, although it is not an error to do:

```
<xsl:attribute name="xmlns:xsl" namespace="whatever">http://www.w3.org/1999/XSL/Tran
```

it will not result in a namespace declaration being output.

Adding an attribute to an element replaces any existing attribute of that element with the same expanded-name.

The following are all errors:

- Adding an attribute to an element after children have been added to it; implementations may either signal the error or ignore the attribute.
- Adding an attribute to a node that is not an element; implementations may either signal the error or ignore the attribute.
- Creating nodes other than text nodes during the instantiation of the content of the `xsl:attribute` element; implementations may either signal the error or ignore the offending nodes.

  NOTE: When an `xsl:attribute` contains a text node with a newline, then the XML output must contain a character reference. For example,

  ```
  <xsl:attribute name="a">x
  y</xsl:attribute>
  ```

  will result in the output

```
a="x&#xA;y"
```

(or with any equivalent character reference). The XML output cannot be

```
a="x
y"
```

This is because XML 1.0 requires newline characters in attribute values to be normalized into spaces but requires character references to newline characters not to be normalized. The attribute values in the data model represent the attribute value after normalization. If a newline occurring in an attribute value in the tree was output as a newline character rather than as character reference, then the attribute value in the tree created by reparsing the XML would contain a space not a newline, which would mean that the tree had not been output correctly.

7.1.4 Named Attribute Sets

```
<!-- Category: top-level-element -->
<xsl:attribute-set
  name = qname
  use-attribute-sets = qnames>
  <!-- Content: xsl:attribute* -->
</xsl:attribute-set>
```

The `xsl:attribute-set` element defines a named set of attributes. The `name` attribute specifies the name of the attribute set. The value of the `name` attribute is a QName, which is expanded as described in [**2.4 Qualified Names**]. The content of the `xsl:attribute-set` element consists of zero or more `xsl:attribute` elements that specify the attributes in the set.

Attribute sets are used by specifying a `use-attribute-sets` attribute on `xsl:element`, `xsl:copy` (see [**7.5 Copying**]) or `xsl:attribute-set` elements. The value of the `use-attribute-sets` attribute is a whitespace-separated list of names of attribute sets. Each name is specified as a QName, which is expanded as described in [**2.4 Qualified Names**]. Specifying a `use-attribute-sets` attribute is equivalent to adding `xsl:attribute` elements for each of the attributes in each of the named attribute sets to the beginning of the content of the element with the `use-attribute-sets` attribute, in the same order in which the names of the attribute sets are specified in the `use-attribute-sets` attribute. It is an error if use of `use-attribute-sets` attributes on `xsl:attribute-set` elements causes an attribute set to directly or indirectly use itself.

Attribute sets can also be used by specifying an `xsl:use-attribute-sets` attribute on a literal result element. The value of the `xsl:use-attribute-sets` attribute is a whitespace-separated list of names of attribute sets. The `xsl:use-attribute-sets` attribute has the same effect as the `use-attribute-sets` attribute on `xsl:element` with the additional rule that attributes specified on the literal result element itself are treated as if they were specified by `xsl:attribute` elements before any actual `xsl:attribute` elements but after any `xsl:attribute` elements implied by the `xsl:use-attribute-sets` attribute. Thus, for a literal result element, attributes from attribute sets named in an `xsl:use-attribute-sets` attribute will be added first, in the order listed in the attribute; next, attributes specified on the literal result element will be added; finally, any attributes specified by `xsl:attribute` elements will be added. Since adding an attribute to an element replaces any existing attribute of that element with the same name, this means that attributes specified in attribute sets can be overridden by attributes specified on the literal result element itself.

The template within each `xsl:attribute` element in an `xsl:attribute-set` element is instantiated each time the attribute set is used; it is instantiated using the same current node and current node list as is used for instantiating the element bearing the `use-attribute-sets` or `xsl:use-attribute-sets` attribute. However, it is the position in the stylesheet of the `xsl:attribute` element rather than of the element bearing the `use-attribute-sets` or `xsl:use-attribute-sets` attribute that determines which variable bindings are visible (see [**11 Variables and Parameters**]); thus, only variables and parameters declared by top-level `xsl:variable` and `xsl:param` elements are visible.

The following example creates a named attribute set `title-style` and uses it in a template rule.

```
<xsl:template match="chapter/heading">
  <fo:block quadding="start" xsl:use-attribute-sets="title-style">
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>

<xsl:attribute-set name="title-style">
  <xsl:attribute name="font-size">12pt</xsl:attribute>
  <xsl:attribute name="font-weight">bold</xsl:attribute>
</xsl:attribute-set>
```

Multiple definitions of an attribute set with the same expanded-name are merged. An attribute from a definition that has higher import precedence takes precedence over an attribute from a definition that has lower import precedence. It is an error if there are two attribute sets that have the same expanded-name and equal import precedence and that both contain the same attribute, unless there is a definition of the attribute set with higher import precedence that also contains the attribute. An XSLT processor may signal the error; if it does not signal the error, it must recover by choosing from amongst the definitions that specify the attribute that have the highest import precedence the one that was specified last in the stylesheet. Where the attributes in an attribute set were specified is relevant only in merging the attributes into the attribute set; it makes no difference when the attribute set is used.

7.2 Creating Text

A template can also contain text nodes. Each text node in a template remaining after whitespace has been stripped as specified in [**3.4 Whitespace Stripping**] will create a text node with the same string-value in the result tree. Adjacent text nodes in the result tree are automatically merged.

Note that text is processed at the tree level. Thus, markup of `<` in a template will be represented in the stylesheet tree by a text node that includes the character <. This will create a text node in the result tree that contains a < character, which will be represented by the markup `<` (or an equivalent character reference) when the result tree is externalized as an XML document (unless output escaping is disabled as described in [**16.4 Disabling Output Escaping**]).

```
<!-- Category: instruction -->
<xsl:text
  disable-output-escaping = "yes" | "no">
  <!-- Content: #PCDATA -->
</xsl:text>
```

Literal data characters may also be wrapped in an `xsl:text` element. This wrapping may change what whitespace characters are stripped (see [**3.4 Whitespace Stripping**]) but does not affect how the characters are handled by the XSLT processor thereafter.

> **NOTE:** The `xml:lang` and `xml:space` attributes are not treated specially by XSLT. In particular,
>
> - it is the responsibility of the stylesheet author explicitly to generate any `xml:lang` or `xml:space` attributes that are needed in the result;
> - specifying an `xml:lang` or `xml:space` attribute on an element in the XSLT namespace will not cause any `xml:lang` or `xml:space` attributes to appear in the result.

7.3 Creating Processing Instructions

```
<!-- Category: instruction -->
<xsl:processing-instruction
  name = { ncname }>
  <!-- Content: template -->
</xsl:processing-instruction>
```

The `xsl:processing-instruction` element is instantiated to create a processing instruction node. The content of the `xsl:processing-instruction` element is a template for the string-value of the processing instruction node. The `xsl:processing-instruction` element has a required `name` attribute that specifies the name of the processing instruction node. The value of the `name` attribute is interpreted as an attribute value template.

For example, this

```
<xsl:processing-instruction name="xml-stylesheet">href="book.css" type="text/css"</x
```

would create the processing instruction

```
<?xml-stylesheet href="book.css" type="text/css"?>
```

It is an error if the string that results from instantiating the `name` attribute is not both an NCName and a PITarget. An XSLT processor may signal the error; if it does not signal the error, it must recover by not adding the processing instruction to the result tree.

> **NOTE:** This means that `xsl:processing-instruction` cannot be used to output an XML declaration. The `xsl:output` element should be used instead (see [**16 Output**]).

It is an error if instantiating the content of `xsl:processing-instruction` creates nodes other than text nodes. An XSLT processor may signal the error; if it does not signal the error, it must recover by ignoring the offending nodes together with their content.

It is an error if the result of instantiating the content of the `xsl:processing-instruction` contains the string `?>`. An XSLT processor may signal the error; if it does not signal the error, it must recover by inserting a space after any occurrence of `?` that is followed by a `>`.

7.4 Creating Comments

```
<!-- Category: instruction -->
<xsl:comment>
  <!-- Content: template -->
</xsl:comment>
```

The `xsl:comment` element is instantiated to create a comment node in the result tree. The content of the `xsl:comment` element is a template for the string-value of the comment node.

For example, this

```
<xsl:comment>This file is automatically generated. Do not edit!</xsl:comment>
```

would create the comment

```
<!--This file is automatically generated. Do not edit!-->
```

It is an error if instantiating the content of `xsl:comment` creates nodes other than text nodes. An XSLT processor may signal the error; if it does not signal the error, it must recover by ignoring the offending nodes together with their content.

It is an error if the result of instantiating the content of the `xsl:comment` contains the string `--` or ends with `-`. An XSLT processor may signal the error; if it does not signal the error, it must recover by inserting a space after any occurrence of `-` that is followed by another `-` or that ends the comment.

7.5 Copying

```
<!-- Category: instruction -->
<xsl:copy
  use-attribute-sets = qnames>
  <!-- Content: template -->
</xsl:copy>
```

The `xsl:copy` element provides an easy way of copying the current node. Instantiating the `xsl:copy` element creates a copy of the current node. The namespace nodes of the current node are automatically copied as well, but the attributes and children of the node are not automatically copied. The content of the `xsl:copy` element is a template for the attributes and children of the created node; the content is instantiated only for nodes of types that can have attributes or children (i.e. root nodes and element nodes).

The `xsl:copy` element may have a `use-attribute-sets` attribute (see [**7.1.4 Named Attribute Sets**]). This is used only when copying element nodes.

The root node is treated specially because the root node of the result tree is created implicitly. When the current node is the root node, `xsl:copy` will not create a root node, but will just use the content template.

For example, the identity transformation can be written using `xsl:copy` as follows:

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

When the current node is an attribute, then if it would be an error to use `xsl:attribute` to create an attribute with the same name as the current node, then it is also an error to use `xsl:copy` (see [**7.1.3 Creating Attributes with `xsl:attribute`**]).

The following example shows how `xml:lang` attributes can be easily copied through from source to result. If a stylesheet defines the following named template:

```
<xsl:template name="apply-templates-copy-lang">
 <xsl:for-each select="@xml:lang">
   <xsl:copy/>
 </xsl:for-each>
 <xsl:apply-templates/>
</xsl:template>
```

then it can simply do

```
<xsl:call-template name="apply-templates-copy-lang"/>
```

instead of

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

when it wants to copy the `xml:lang` attribute.

7.6 Computing Generated Text

Within a template, the `xsl:value-of` element can be used to compute generated text, for example by extracting text from the source tree or by inserting the value of a variable. The `xsl:value-of` element does this with an expression that is specified as the value of the `select` attribute. Expressions can also be used inside attribute values of literal result elements by enclosing the expression in curly braces (`{}`).

7.6.1 Generating Text with `xsl:value-of`

```
<!-- Category: instruction -->
<xsl:value-of
  select = string-expression
  disable-output-escaping = "yes" | "no" />
```

The `xsl:value-of` element is instantiated to create a text node in the result tree. The required `select` attribute is an expression; this expression is evaluated and the resulting object is converted to a string as if by a call to the **string** function. The string specifies the string-value of the created text node. If the string is empty, no text node will be created. The created text node will be merged with any adjacent text nodes.

The `xsl:copy-of` element can be used to copy a node-set over to the result tree without converting it to a string. See [**11.3 Using Values of Variables and Parameters with `xsl:copy-of`**].

For example, the following creates an HTML paragraph from a `person` element with `given-name` and `family-name` attributes. The paragraph will contain the value of the `given-name` attribute of the current node followed by a space and the value of the `family-name` attribute of the current node.

```
<xsl:template match="person">
  <p>
   <xsl:value-of select="@given-name"/>
   <xsl:text> </xsl:text>
   <xsl:value-of select="@family-name"/>
  </p>
</xsl:template>
```

For another example, the following creates an HTML paragraph from a `person` element with `given-name` and `family-name` children elements. The paragraph will contain the string-value of the first `given-name` child element of the current node followed by a space and the string-value of the first `family-name` child element of the current node.

```
<xsl:template match="person">
  <p>
   <xsl:value-of select="given-name"/>
   <xsl:text> </xsl:text>
   <xsl:value-of select="family-name"/>
  </p>
</xsl:template>
```

The following precedes each `procedure` element with a paragraph containing the security level of the procedure. It assumes that the security level that applies to a procedure is determined by a `security` attribute on the procedure element or on an ancestor element of the procedure. It also assumes that if more than one such element has a `security` attribute then the security level is determined by the element that is closest to the procedure.

```
<xsl:template match="procedure">
  <fo:block>
    <xsl:value-of select="ancestor-or-self::*[@security][1]/@security"/>
  </fo:block>
  <xsl:apply-templates/>
</xsl:template>
```

7.6.2 Attribute Value Templates

In an attribute value that is interpreted as an **attribute value template**, such as an attribute of a literal result element, an expression can be used by surrounding the expression with curly braces (`{}`). The attribute value template is instantiated by replacing the expression together with surrounding curly braces by the result of evaluating the expression and converting the resulting object to a string as if by a call to the **string** function. Curly braces are not recognized in an attribute value in an XSLT stylesheet unless the attribute is specifically stated to be one that is interpreted as an attribute value template; in an element syntax summary, the value of such attributes is surrounded by curly braces.

> **NOTE:** Not all attributes are interpreted as attribute value templates. Attributes whose value is an expression or pattern, attributes of top-level elements and attributes that refer to named XSLT objects are not interpreted as attribute value templates. In addition, `xmlns` attributes are not interpreted as attribute value templates; it would not be conformant with the XML

Namespaces Recommendation to do this.

The following example creates an `img` result element from a `photograph` element in the source; the value of the `src` attribute of the `img` element is computed from the value of the `image-dir` variable and the string-value of the `href` child of the `photograph` element; the value of the `width` attribute of the `img` element is computed from the value of the `width` attribute of the `size` child of the `photograph` element:

```
<xsl:variable name="image-dir">/images</xsl:variable>

<xsl:template match="photograph">
<img src="{$image-dir}/{href}" width="{size/@width}"/>
</xsl:template>
```

With this source

```
<photograph>
  <href>headquarters.jpg</href>
  <size width="300"/>
</photograph>
```

the result would be

```
<img src="/images/headquarters.jpg" width="300"/>
```

When an attribute value template is instantiated, a double left or right curly brace outside an expression will be replaced by a single curly brace. It is an error if a right curly brace occurs in an attribute value template outside an expression without being followed by a second right curly brace. A right curly brace inside a Literal in an expression is not recognized as terminating the expression.

Curly braces are *not* recognized recursively inside expressions. For example:

```
<a href="#{id({@ref})/title}">
```

is *not* allowed. Instead, use simply:

```
<a href="#{id(@ref)/title}">
```

7.7 Numbering

```
<!-- Category: instruction -->
<xsl:number
  level = "single" | "multiple" | "any"
  count = pattern
  from = pattern
  value = number-expression
  format = { string }
  lang = { nmtoken }
  letter-value = { "alphabetic" | "traditional" }
  grouping-separator = { char }
  grouping-size = { number } />
```

The `xsl:number` element is used to insert a formatted number into the result tree. The number to be inserted may be specified by an expression. The `value` attribute contains an expression. The expression is evaluated and the resulting object is converted to a number as if by a call to the **number** function. The number is rounded to an integer and then converted to a string using the attributes specified in [**7.7.1 Number to String Conversion Attributes**]; in this context, the value of each of these attributes is interpreted as an attribute value template. After conversion, the resulting string is inserted in the result tree. For example, the following example numbers a sorted list:

```
<xsl:template match="items">
  <xsl:for-each select="item">
    <xsl:sort select="."/>
    <p>
      <xsl:number value="position()" format="1. "/>
      <xsl:value-of select="."/>
    </p>
  </xsl:for-each>
</xsl:template>
```

If no `value` attribute is specified, then the `xsl:number` element inserts a number based on the position of the current node in the source tree. The following attributes control how the current node is to be numbered:

- The `level` attribute specifies what levels of the source tree should be considered; it has the values `single`, `multiple` or `any`. The default is `single`.

- The `count` attribute is a pattern that specifies what nodes should be counted at those levels. If `count` attribute is not specified, then it defaults to the pattern that matches any node with the same node type as the current node and, if the current node has an expanded-name, with the same expanded-name as the current node.

- The `from` attribute is a pattern that specifies where counting starts.

In addition, the attributes specified in [**7.7.1 Number to String Conversion Attributes**] are used for number to string conversion, as in the case when the `value` attribute is specified.

The `xsl:number` element first constructs a list of positive integers using the `level`, `count` and `from` attributes:

- When `level="single"`, it goes up to the first node in the ancestor-or-self axis that matches the `count` pattern, and constructs a list of length one containing one plus the number of preceding siblings of that ancestor that match the `count` pattern. If there is no such ancestor, it constructs an empty list. If the `from` attribute is specified, then the only ancestors that are searched are those that are descendants of the nearest ancestor that matches the `from` pattern. Preceding siblings has the same meaning here as with the `preceding-sibling` axis.

- When `level="multiple"`, it constructs a list of all ancestors of the current node in document order followed by the element itself; it then selects from the list those nodes that match the `count` pattern; it then maps each node in the list to one plus the number of preceding siblings of that node that match the `count` pattern. If the `from` attribute is specified, then the only ancestors that are searched are those that are descendants of the nearest ancestor that matches the `from` pattern. Preceding siblings has the same meaning here as with the `preceding-sibling` axis.

- When `level="any"`, it constructs a list of length one containing the number of nodes that match the `count` pattern and belong to the set containing the current node and all nodes at any level of the document that are before the current node in document order, excluding any namespace and attribute nodes (in other words the union of the members of the `preceding` and `ancestor-or-self` axes). If the `from` attribute is specified, then only nodes after the first node before the current node that match the `from` pattern are considered.

The list of numbers is then converted into a string using the attributes specified in [**7.7.1 Number to String Conversion Attributes**]; in this context, the value of each of these attributes is interpreted as an attribute value template. After conversion, the resulting string is inserted in the result tree.

The following would number the items in an ordered list:

```
<xsl:template match="ol/item">
  <fo:block>
    <xsl:number/><xsl:text>. </xsl:text><xsl:apply-templates/>
  </fo:block>
<xsl:template>
```

The following two rules would number `title` elements. This is intended for a document that contains a sequence of chapters followed by a sequence of appendices, where both chapters and appendices contain sections, which in turn contain subsections. Chapters are numbered 1, 2, 3; appendices are numbered A, B, C; sections in chapters are numbered 1.1, 1.2, 1.3; sections in appendices are numbered A.1, A.2, A.3.

```
<xsl:template match="title">
  <fo:block>
     <xsl:number level="multiple"
                 count="chapter|section|subsection"
                 format="1.1 "/>
     <xsl:apply-templates/>
  </fo:block>
</xsl:template>

<xsl:template match="appendix//title" priority="1">
  <fo:block>
     <xsl:number level="multiple"
                 count="appendix|section|subsection"
                 format="A.1 "/>
     <xsl:apply-templates/>
  </fo:block>
</xsl:template>
```

The following example numbers notes sequentially within a chapter:

```
<xsl:template match="note">
  <fo:block>
     <xsl:number level="any" from="chapter" format="(1) "/>
     <xsl:apply-templates/>
  </fo:block>
</xsl:template>
```

The following example would number `H4` elements in HTML with a three-part label:

```
<xsl:template match="H4">
 <fo:block>
```

```
    <xsl:number level="any" from="H1" count="H2"/>
    <xsl:text>.</xsl:text>
    <xsl:number level="any" from="H2" count="H3"/>
    <xsl:text>.</xsl:text>
    <xsl:number level="any" from="H3" count="H4"/>
    <xsl:text> </xsl:text>
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>
```

7.7.1 Number to String Conversion Attributes

The following attributes are used to control conversion of a list of numbers into a string. The numbers are integers greater than 0. The attributes are all optional.

The main attribute is `format`. The default value for the `format` attribute is `1`. The `format` attribute is split into a sequence of tokens where each token is a maximal sequence of alphanumeric characters or a maximal sequence of non-alphanumeric characters. Alphanumeric means any character that has a Unicode category of Nd, Nl, No, Lu, Ll, Lt, Lm or Lo. The alphanumeric tokens (format tokens) specify the format to be used for each number in the list. If the first token is a non-alphanumeric token, then the constructed string will start with that token; if the last token is non-alphanumeric token, then the constructed string will end with that token. Non-alphanumeric tokens that occur between two format tokens are separator tokens that are used to join numbers in the list. The *n*th format token will be used to format the *n*th number in the list. If there are more numbers than format tokens, then the last format token will be used to format remaining numbers. If there are no format tokens, then a format token of `1` is used to format all numbers. The format token specifies the string to be used to represent the number 1. Each number after the first will be separated from the preceding number by the separator token preceding the format token used to format that number, or, if there are no separator tokens, then by `.` (a period character).

Format tokens are a superset of the allowed values for the `type` attribute for the `OL` element in HTML 4.0 and are interpreted as follows:

- Any token where the last character has a decimal digit value of 1 (as specified in the Unicode character property database), and the Unicode value of preceding characters is one less than the Unicode value of the last character generates a decimal representation of the number where each number is at least as long as the format token. Thus, a format token `1` generates the sequence `1 2 ... 10 11 12 ...`, and a format token `01` generates the sequence `01 02 ... 09 10 11 12 ... 99 100 101`.
- A format token `A` generates the sequence `A B C ... Z AA AB AC...`.
- A format token `a` generates the sequence `a b c ... z aa ab ac...`.
- A format token `i` generates the sequence `i ii iii iv v vi vii viii ix x ...`.
- A format token `I` generates the sequence `I II III IV V VI VII VIII IX X ...`.
- Any other format token indicates a numbering sequence that starts with that token. If an implementation does not support a numbering sequence that starts with that token, it must use a

format token of `1`.

When numbering with an alphabetic sequence, the `lang` attribute specifies which language's alphabet is to be used; it has the same range of values as `xml:lang` [XML]; if no `lang` value is specified, the language should be determined from the system environment. Implementers should document for which languages they support numbering.

> **NOTE:** Implementers should not make any assumptions about how numbering works in particular languages and should properly research the languages that they wish to support. The numbering conventions of many languages are very different from English.

The `letter-value` attribute disambiguates between numbering sequences that use letters. In many languages there are two commonly used numbering sequences that use letters. One numbering sequence assigns numeric values to letters in alphabetic sequence, and the other assigns numeric values to each letter in some other manner traditional in that language. In English, these would correspond to the numbering sequences specified by the format tokens `a` and `i`. In some languages, the first member of each sequence is the same, and so the format token alone would be ambiguous. A value of `alphabetic` specifies the alphabetic sequence; a value of `traditional` specifies the other sequence. If the `letter-value` attribute is not specified, then it is implementation-dependent how any ambiguity is resolved.

> **NOTE:** It is possible for two conforming XSLT processors not to convert a number to exactly the same string. Some XSLT processors may not support some languages. Furthermore, there may be variations possible in the way conversions are performed for any particular language that are not specifiable by the attributes on `xsl:number`. Future versions of XSLT may provide additional attributes to provide control over these variations. Implementations may also use implementation-specific namespaced attributes on `xsl:number` for this.

The `grouping-separator` attribute gives the separator used as a grouping (e.g. thousands) separator in decimal numbering sequences, and the optional `grouping-size` specifies the size (normally 3) of the grouping. For example, `grouping-separator=","` and `grouping-size="3"` would produce numbers of the form `1,000,000`. If only one of the `grouping-separator` and `grouping-size` attributes is specified, then it is ignored.

Here are some examples of conversion specifications:

- `format="ア"` specifies Katakana numbering
- `format="イ"` specifies Katakana numbering in the "iroha" order
- `format="๑"` specifies numbering with Thai digits
- `format="א" letter-value="traditional"` specifies "traditional" Hebrew numbering
- `format="ა" letter-value="traditional"` specifies Georgian numbering
- `format="α" letter-value="traditional"` specifies "classical" Greek numbering

- `format="а" letter-value="traditional"` specifies Old Slavic numbering

8 Repetition

```
<!-- Category: instruction -->
<xsl:for-each
  select = node-set-expression>
  <!-- Content: (xsl:sort*, template) -->
</xsl:for-each>
```

When the result has a known regular structure, it is useful to be able to specify directly the template for selected nodes. The `xsl:for-each` instruction contains a template, which is instantiated for each node selected by the expression specified by the `select` attribute. The `select` attribute is required. The expression must evaluate to a node-set. The template is instantiated with the selected node as the current node, and with a list of all of the selected nodes as the current node list. The nodes are processed in document order, unless a sorting specification is present (see [**10 Sorting**]).

For example, given an XML document with this structure

```
<customers>
  <customer>
    <name>...</name>
    <order>...</order>
    <order>...</order>
  </customer>
  <customer>
    <name>...</name>
    <order>...</order>
    <order>...</order>
  </customer>
</customers>
```

the following would create an HTML document containing a table with a row for each `customer` element

```
<xsl:template match="/">
  <html>
    <head>
      <title>Customers</title>
    </head>
    <body>
      <table>
        <tbody>
          <xsl:for-each select="customers/customer">
            <tr>
              <th>
                <xsl:apply-templates select="name"/>
              </th>
              <xsl:for-each select="order">
                <td>
                  <xsl:apply-templates/>
                </td>
              </xsl:for-each>
            </tr>
          </xsl:for-each>
```

```
      </tbody>
    </table>
  </body>
</html>
</xsl:template>
```

9 Conditional Processing

There are two instructions in XSLT that support conditional processing in a template: `xsl:if` and `xsl:choose`. The `xsl:if` instruction provides simple if-then conditionality; the `xsl:choose` instruction supports selection of one choice when there are several possibilities.

9.1 Conditional Processing with `xsl:if`

```
<!-- Category: instruction -->
<xsl:if
  test = boolean-expression>
  <!-- Content: template -->
</xsl:if>
```

The `xsl:if` element has a `test` attribute, which specifies an expression. The content is a template. The expression is evaluated and the resulting object is converted to a boolean as if by a call to the **boolean** function. If the result is true, then the content template is instantiated; otherwise, nothing is created. In the following example, the names in a group of names are formatted as a comma separated list:

```
<xsl:template match="namelist/name">
  <xsl:apply-templates/>
  <xsl:if test="not(position()=last())">, </xsl:if>
</xsl:template>
```

The following colors every other table row yellow:

```
<xsl:template match="item">
  <tr>
    <xsl:if test="position() mod 2 = 0">
       <xsl:attribute name="bgcolor">yellow</xsl:attribute>
    </xsl:if>
    <xsl:apply-templates/>
  </tr>
</xsl:template>
```

9.2 Conditional Processing with `xsl:choose`

```
<!-- Category: instruction -->
<xsl:choose>
  <!-- Content: (xsl:when+, xsl:otherwise?) -->
</xsl:choose>
```

```
<xsl:when
  test = boolean-expression>
  <!-- Content: template -->
</xsl:when>
```

```
<xsl:otherwise>
```

```
  <!-- Content: template -->
</xsl:otherwise>
```

The `xsl:choose` element selects one among a number of possible alternatives. It consists of a sequence of `xsl:when` elements followed by an optional `xsl:otherwise` element. Each `xsl:when` element has a single attribute, `test`, which specifies an expression. The content of the `xsl:when` and `xsl:otherwise` elements is a template. When an `xsl:choose` element is processed, each of the `xsl:when` elements is tested in turn, by evaluating the expression and converting the resulting object to a boolean as if by a call to the **boolean** function. The content of the first, and only the first, `xsl:when` element whose test is true is instantiated. If no `xsl:when` is true, the content of the `xsl:otherwise` element is instantiated. If no `xsl:when` element is true, and no `xsl:otherwise` element is present, nothing is created.

The following example enumerates items in an ordered list using arabic numerals, letters, or roman numerals depending on the depth to which the ordered lists are nested.

```
<xsl:template match="orderedlist/listitem">
  <fo:list-item indent-start='2pi'>
    <fo:list-item-label>
      <xsl:variable name="level"
                    select="count(ancestor::orderedlist) mod 3"/>
      <xsl:choose>
        <xsl:when test='$level=1'>
          <xsl:number format="i"/>
        </xsl:when>
        <xsl:when test='$level=2'>
          <xsl:number format="a"/>
        </xsl:when>
        <xsl:otherwise>
          <xsl:number format="1"/>
        </xsl:otherwise>
      </xsl:choose>
      <xsl:text>. </xsl:text>
    </fo:list-item-label>
    <fo:list-item-body>
      <xsl:apply-templates/>
    </fo:list-item-body>
  </fo:list-item>
</xsl:template>
```

10 Sorting

```
<xsl:sort
  select = string-expression
  lang = { nmtoken }
  data-type = { "text" | "number" }
  order = { "ascending" | "descending" }
  case-order = { "upper-first" | "lower-first" } />
```

Sorting is specified by adding `xsl:sort` elements as children of an `xsl:apply-templates` or `xsl:for-each` element. The first `xsl:sort` child specifies the primary sort key, the second `xsl:sort` child specifies the secondary sort key and so on. When an `xsl:apply-templates` or `xsl:for-each` element has one or more `xsl:sort` children, then instead of processing the selected nodes in document order, it sorts the nodes according to the specified sort keys and then processes them in sorted order. When used in `xsl:for-each`, `xsl:sort` elements must occur first. When a template is instantiated by `xsl:apply-templates` and `xsl:for-each`, the current node list list consists of the complete list of nodes being processed in sorted order.

`xsl:sort` has a `select` attribute whose value is an expression. For each node to be processed, the expression is evaluated with that node as the current node. The resulting object is converted to a string as if by a call to the **string** function; this string is used as the sort key for that node. The default value of the `select` attribute is `.`, which will cause the string-value of the current node to be used as the sort key.

This string serves as a sort key for the node. The following optional attributes on `xsl:sort` control how the list of sort keys are sorted; the values of all of these attributes are interpreted as attribute value templates.

- `order` specifies whether the strings should be sorted in ascending or descending order; `ascending` specifies ascending order; `descending` specifies descending order; the default is `ascending`
- `lang` specifies the language of the sort keys; it has the same range of values as `xml:lang` [XML]; if no `lang` value is specified, the language should be determined from the system environment
- `data-type` specifies the data type of the strings; the following values are allowed:
  - `text` specifies that the sort keys should be sorted lexicographically in the culturally correct manner for the language specified by `lang`
  - `number` specifies that the sort keys should be converted to numbers and then sorted according to the numeric value; leading and trailing whitespace is stripped from the sort key and the sort key is then converted to a number as if by a call to the **number** function; the `lang` attribute is ignored

  The default value is `text`.

  > **NOTE:** The XSL Working Group plans that future versions of XSLT will leverage XML Schemas to define further values for this attribute.

- `case-order` has the value `upper-first` or `lower-first`; this applies when `data-type="text"`, and specifies that upper-case letters should sort before lower-case letters or vice-versa respectively. For example, if `lang="en"`, then `A a B b` are sorted with `case-order="upper-first"` and `a A b B` are sorted with `case-order="lower-first"`. The default value is language dependent.

  NOTE: It is possible for two conforming XSLT processors not to sort exactly the same. Some XSLT processors may not support some languages. Furthermore, there may be variations possible in the sorting of any particular language that are not specified by the attributes on `xsl:sort`, for example, whether Hiragana or Katakana is sorted first in Japanese. Future versions of XSLT may provide additional attributes to provide control over these variations. Implementations may also use implementation-specific namespaced attributes on `xsl:sort` for this.

  NOTE: It is recommended that implementers consult [UNICODE TR10] for information on internationalized sorting.

The sort must be stable: in the sorted list of nodes, any sub list that has sort keys that all compare equal must be in document order.

For example, suppose an employee database has the form

```
<employees>
  <employee>
    <name>
      <given>James</given>
      <family>Clark</family>
    </name>
    ...
  </employee>
</employees>
```

Then a list of employees sorted by name could be generated using:

```
<xsl:template match="employees">
  <ul>
    <xsl:apply-templates select="employee">
      <xsl:sort select="name/family"/>
      <xsl:sort select="name/given"/>
    </xsl:apply-templates>
  </ul>
</xsl:template>

<xsl:template match="employee">
  <li>
    <xsl:value-of select="name/given"/>
    <xsl:text> </xsl:text>
    <xsl:value-of select="name/family"/>
  </li>
</xsl:template>
```

11 Variables and Parameters

```
<!-- Category: top-level-element -->
<!-- Category: instruction -->
<xsl:variable
  name = qname
  select = expression>
  <!-- Content: template -->
</xsl:variable>

<!-- Category: top-level-element -->
<xsl:param
  name = qname
  select = expression>
  <!-- Content: template -->
</xsl:param>
```

A variable is a name that may be bound to a value. The value to which a variable is bound (the **value** of the variable) can be an object of any of the types that can be returned by expressions. There are two elements that can be used to bind variables: `xsl:variable` and `xsl:param`. The difference is that the value specified on the `xsl:param` variable is only a default value for the binding; when the template or stylesheet within which the `xsl:param` element occurs is invoked, parameters may be passed that are used in place of the default values.

Both `xsl:variable` and `xsl:param` have a required `name` attribute, which specifies the name of the variable. The value of the `name` attribute is a QName, which is expanded as described in [**2.4 Qualified Names**].

For any use of these variable-binding elements, there is a region of the stylesheet tree within which the binding is visible; within this region, any binding of the variable that was visible on the variable-binding element itself is hidden. Thus, only the innermost binding of a variable is visible. The set of variable bindings in scope for an expression consists of those bindings that are visible at the point in the stylesheet where the expression occurs.

11.1 Result Tree Fragments

Variables introduce an additional data-type into the expression language. This additional data type is called **result tree fragment**. A variable may be bound to a result tree fragment instead of one of the four basic XPath data-types (string, number, boolean, node-set). A result tree fragment represents a fragment of the result tree. A result tree fragment is treated equivalently to a node-set that contains just a single root node. However, the operations permitted on a result tree fragment are a subset of those permitted on a node-set. An operation is permitted on a result tree fragment only if that operation would be permitted on a string (the operation on the string may involve first converting the string to a number or boolean). In particular, it is not permitted to use the `/`, `//`, and `[]` operators on result tree fragments. When a permitted operation is performed on a result tree fragment, it is performed exactly as it would be on the equivalent node-set.

When a result tree fragment is copied into the result tree (see [**11.3 Using Values of Variables and Parameters with `xsl:copy-of`**]), then all the nodes that are children of the root node in the equivalent node-set are added in sequence to the result tree.

Expressions can only return values of type result tree fragment by referencing variables of type result tree fragment or calling extension functions that return a result tree fragment or getting a system property whose value is a result tree fragment.

11.2 Values of Variables and Parameters

A variable-binding element can specify the value of the variable in three alternative ways.

- If the variable-binding element has a `select` attribute, then the value of the attribute must be an expression and the value of the variable is the object that results from evaluating the expression. In this case, the content must be empty.

- If the variable-binding element does not have a `select` attribute and has non-empty content (i.e. the variable-binding element has one or more child nodes), then the content of the variable-binding element specifies the value. The content of the variable-binding element is a template, which is instantiated to give the value of the variable. The value is a result tree fragment equivalent to a node-set containing just a single root node having as children the sequence of nodes produced by instantiating the template. The base URI of the nodes in the result tree fragment is the base URI of the variable-binding element.

It is an error if a member of the sequence of nodes created by instantiating the template is an attribute node or a namespace node, since a root node cannot have an attribute node or a namespace node as a child. An XSLT processor may signal the error; if it does not signal the error, it must recover by not adding the attribute node or namespace node.

- If the variable-binding element has empty content and does not have a `select` attribute, then the value of the variable is an empty string. Thus

  ```
  <xsl:variable name="x"/>
  ```

  is equivalent to

  ```
  <xsl:variable name="x" select="''"/>
  ```

  NOTE: When a variable is used to select nodes by position, be careful not to do:

  ```
  <xsl:variable name="n">2</xsl:variable>
  ...
  <xsl:value-of select="item[$n]"/>
  ```

  This will output the value of the first item element, because the variable `n` will be bound to a result tree fragment, not a number. Instead, do either

  ```
  <xsl:variable name="n" select="2"/>
  ...
  <xsl:value-of select="item[$n]"/>
  ```

  or

  ```
  <xsl:variable name="n">2</xsl:variable>
  ...
  <xsl:value-of select="item[position()=$n]"/>
  ```

  NOTE: One convenient way to specify the empty node-set as the default value of a parameter is:

  ```
  <xsl:param name="x" select="/.."/>
  ```

11.3 Using Values of Variables and Parameters with `xsl:copy-of`

```
<!-- Category: instruction -->
<xsl:copy-of
  select = expression />
```

The `xsl:copy-of` element can be used to insert a result tree fragment into the result tree, without first converting it to a string as `xsl:value-of` does (see [**7.6.1 Generating Text with `xsl:value-of`**]). The required `select` attribute contains an expression. When the result of evaluating the expression is a result tree fragment, the complete fragment is copied into the result tree. When the result is a node-set, all the nodes in the set together with their content are copied in document order over into the result tree; a root node is copied by copying its children. When it is of any other type, the result is converted to a string and then inserted into the result tree, as with `xsl:value-of`.

11.4 Top-level Variables and Parameters

Both `xsl:variable` and `xsl:param` are allowed as top-level elements. A top-level variable-binding element declares a global variable that is visible everywhere. A top-level `xsl:param` element declares a parameter to the stylesheet; XSLT does not define the mechanism by which parameters are passed to the stylesheet. It is an error if a stylesheet contains more than one binding of a top-level variable with the same name and same import precedence. At the top-level, the expression or template specifying the variable value is evaluated with the same context as that used to process the root node of the source document: the current node is the root node of the source document and the current node list is a list containing just the root node of the source document. If the template or expression specifying the value of a global variable *x* references a global variable *y*, then the value for *y* must be computed before the value of *x*. It is an error if it is impossible to do this for all global variable definitions; in other words, it is an error if the definitions are circular.

This example declares a global variable `para-font-size`, which it references in an attribute value template.

```
<xsl:variable name="para-font-size">12pt</xsl:variable>

<xsl:template match="para">
 <fo:block font-size="{$para-font-size}">
   <xsl:apply-templates/>
 </fo:block>
</xsl:template>
```

11.5 Variables and Parameters within Templates

As well as being allowed at the top-level, both `xsl:variable` and `xsl:param` are also allowed in templates. `xsl:variable` is allowed anywhere within a template that an instruction is allowed. In this case, the binding is visible for all following siblings and their descendants. Note that the binding is not visible for the `xsl:variable` element itself. `xsl:param` is allowed as a child at the beginning of an `xsl:template` element. In this context, the binding is visible for all following siblings and their descendants. Note that the binding is not visible for the `xsl:param` element itself.

A binding **shadows** another binding if the binding occurs at a point where the other binding is visible, and the bindings have the same name. It is an error if a binding established by an `xsl:variable` or `xsl:param` element within a template shadows another binding established by an `xsl:variable` or `xsl:param` element also within the template. It is not an error if a binding established by an `xsl:variable` or `xsl:param` element in a template shadows another binding established by an `xsl:variable` or `xsl:param` top-level element. Thus, the following is an error:

```
<xsl:template name="foo">
<xsl:param name="x" select="1"/>
<xsl:variable name="x" select="2"/>
</xsl:template>
```

However, the following is allowed:

```
<xsl:param name="x" select="1"/>
<xsl:template name="foo">
<xsl:variable name="x" select="2"/>
</xsl:template>
```

NOTE: The nearest equivalent in Java to an `xsl:variable` element in a template is a final local variable declaration with an initializer. For example,

```
<xsl:variable name="x" select="'value'"/>
```

has similar semantics to

```
final Object x = "value";
```

XSLT does not provide an equivalent to the Java assignment operator

```
x = "value";
```

because this would make it harder to create an implementation that processes a document other than in a batch-like way, starting at the beginning and continuing through to the end.

11.6 Passing Parameters to Templates

```
<xsl:with-param
  name = qname
  select = expression>
  <!-- Content: template -->
</xsl:with-param>
```

Parameters are passed to templates using the `xsl:with-param` element. The required `name` attribute specifies the name of the parameter (the variable the value of whose binding is to be replaced). The value of the `name` attribute is a QName, which is expanded as described in [**2.4 Qualified Names**]. `xsl:with-param` is allowed within both `xsl:call-template` and `xsl:apply-templates`. The value of the parameter is specified in the same way as for `xsl:variable` and `xsl:param`. The current node and current node list used for computing the value specified by `xsl:with-param` element is the same as that used for the `xsl:apply-templates` or `xsl:call-template` element within which it occurs. It is not an error to pass a parameter *x* to a template that does not have an `xsl:param` element for *x*; the parameter is simply ignored.

This example defines a named template for a `numbered-block` with an argument to control the format of the number.

```
<xsl:template name="numbered-block">
  <xsl:param name="format">1. </xsl:param>
  <fo:block>
    <xsl:number format="{$format}"/>
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>

<xsl:template match="ol//ol/li">
  <xsl:call-template name="numbered-block">
    <xsl:with-param name="format">a. </xsl:with-param>
  </xsl:call-template>
```

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

12 Additional Functions

This section describes XSLT-specific additions to the core XPath function library. Some of these additional functions also make use of information specified by top-level elements in the stylesheet; this section also describes these elements.

12.1 Multiple Source Documents

Function: *node-set* **document**(*object*, *node-set*?)

The **document** function allows access to XML documents other than the main source document.

When the **document** function has exactly one argument and the argument is a node-set, then the result is the union, for each node in the argument node-set, of the result of calling the **document** function with the first argument being the string-value of the node, and the second argument being a node-set with the node as its only member. When the **document** function has two arguments and the first argument is a node-set, then the result is the union, for each node in the argument node-set, of the result of calling the **document** function with the first argument being the string-value of the node, and with the second argument being the second argument passed to the **document** function.

When the first argument to the **document** function is not a node-set, the first argument is converted to a string as if by a call to the **string** function. This string is treated as a URI reference; the resource identified by the URI is retrieved. The data resulting from the retrieval action is parsed as an XML document and a tree is constructed in accordance with the data model (see [**3 Data Model**]). If there is an error retrieving the resource, then the XSLT processor may signal an error; if it does not signal an error, it must recover by returning an empty node-set. One possible kind of retrieval error is that the XSLT processor does not support the URI scheme used by the URI. An XSLT processor is not required to support any particular URI schemes. The documentation for an XSLT processor should specify which URI schemes the XSLT processor supports.

If the URI reference does not contain a fragment identifier, then a node-set containing just the root node of the document is returned. If the URI reference does contain a fragment identifier, the function returns a node-set containing the nodes in the tree identified by the fragment identifier of the URI reference. The semantics of the fragment identifier is dependent on the media type of the result of retrieving the URI. If there is an error in processing the fragment identifier, the XSLT processor may signal the error; if it does not signal the error, it must recover by returning an empty node-set. Possible errors include:

- The fragment identifier identifies something that cannot be represented by an XSLT node-set (such as a range of characters within a text node).
- The XSLT processor does not support fragment identifiers for the media-type of the retrieval result. An XSLT processor is not required to support any particular media types. The documentation for an XSLT processor should specify for which media types the XSLT processor supports fragment identifiers.

The data resulting from the retrieval action is parsed as an XML document regardless of the media type of the retrieval result; if the top-level media type is `text`, then it is parsed in the same way as if the media type were `text/xml`; otherwise, it is parsed in the same way as if the media type were `application/xml`.

> **NOTE:** Since there is no top-level `xml` media type, data with a media type other than `text/xml` or `application/xml` may in fact be XML.

The URI reference may be relative. The base URI (see [**3.2 Base URI**]) of the node in the second argument node-set that is first in document order is used as the base URI for resolving the relative URI into an absolute URI. If the second argument is omitted, then it defaults to the node in the stylesheet that contains the expression that includes the call to the **document** function. Note that a zero-length URI reference is a reference to the document relative to which the URI reference is being resolved; thus `document("")` refers to the root node of the stylesheet; the tree representation of the stylesheet is exactly the same as if the XML document containing the stylesheet was the initial source document.

Two documents are treated as the same document if they are identified by the same URI. The URI used for the comparison is the absolute URI into which any relative URI was resolved and does not include any fragment identifier. One root node is treated as the same node as another root node if the two nodes are from the same document. Thus, the following expression will always be true:

```
generate-id(document("foo.xml"))=generate-id(document("foo.xml"))
```

The **document** function gives rise to the possibility that a node-set may contain nodes from more than one document. With such a node-set, the relative document order of two nodes in the same document is the normal document order defined by XPath [XPath]. The relative document order of two nodes in different documents is determined by an implementation-dependent ordering of the documents containing the two nodes. There are no constraints on how the implementation orders documents other than that it must do so consistently: an implementation must always use the same order for the same set of documents.

12.2 Keys

Keys provide a way to work with documents that contain an implicit cross-reference structure. The `ID`, `IDREF` and `IDREFS` attribute types in XML provide a mechanism to allow XML documents to make their cross-reference explicit. XSLT supports this through the XPath **id** function. However, this mechanism has a number of limitations:

- ID attributes must be declared as such in the DTD. If an ID attribute is declared as an ID attribute only in the external DTD subset, then it will be recognized as an ID attribute only if the XML processor reads the external DTD subset. However, XML does not require XML processors to read the external DTD, and they may well choose not to do so, especially if the document is declared `standalone="yes"`.

- A document can contain only a single set of unique IDs. There cannot be separate independent sets of unique IDs.

- The ID of an element can only be specified in an attribute; in cannot be specified by the content of the element, or by a child element.

- An ID is constrained to be an XML name. For example, it cannot contain spaces.
- An element can have at most one ID.
- At most one element can have a particular ID.

Because of these limitations XML documents sometimes contain a cross-reference structure that is not explicitly declared by ID/IDREF/IDREFS attributes.

A key is a triple containing:

1. the node which has the key
2. the name of the key (an expanded-name)
3. the value of the key (a string)

A stylesheet declares a set of keys for each document using the `xsl:key` element. When this set of keys contains a member with node *x*, name *y* and value *z*, we say that node *x* has a key with name *y* and value *z*.

Thus, a key is a kind of generalized ID, which is not subject to the same limitations as an XML ID:

- Keys are declared in the stylesheet using `xsl:key` elements.
- A key has a name as well as a value; each key name may be thought of as distinguishing a separate, independent space of identifiers.
- The value of a named key for an element may be specified in any convenient place; for example, in an attribute, in a child element or in content. An XPath expression is used to specify where to find the value for a particular named key.
- The value of a key can be an arbitrary string; it is not constrained to be a name.
- There can be multiple keys in a document with the same node, same key name, but different key values.
- There can be multiple keys in a document with the same key name, same key value, but different nodes.

```
<!-- Category: top-level-element -->
<xsl:key
  name = qname
  match = pattern
  use = node-set-expression />
```

The `xsl:key` element is used to declare keys. The `name` attribute specifies the name of the key. The value of the `name` attribute is a QName, which is expanded as described in [**2.4 Qualified Names**]. The `match` attribute is a Pattern; an `xsl:key` element gives information about the keys of any node that matches the pattern specified in the match attribute. The `use` attribute is an expression specifying the values of the key; the expression is evaluated once for each node that matches the pattern. If the result is a node-set, then for each node in the node-set, the node that matches the pattern has a key of the specified name whose value is the string-value of the node in the node-set; otherwise, the result is converted to a string, and the node that matches the pattern has a key of the specified name with value equal to that string. Thus, a node *x* has a key with name *y* and value *z* if and only if there is an `xsl:key` element such that:

- *x* matches the pattern specified in the `match` attribute of the `xsl:key` element;
- the value of the `name` attribute of the `xsl:key` element is equal to *y*; and
- when the expression specified in the `use` attribute of the `xsl:key` element is evaluated with *x* as the current node and with a node list containing just *x* as the current node list resulting in an object *u*, then either *z* is equal to the result of converting *u* to a string as if by a call to the **string** function, or *u* is a node-set and *z* is equal to the string-value of one or more of the nodes in *u*.

Note also that there may be more than one `xsl:key` element that matches a given node; all of the matching `xsl:key` elements are used, even if they do not have the same import precedence.

It is an error for the value of either the `use` attribute or the `match` attribute to contain a VariableReference.

Function: *node-set* **key**(*string*, *object*)

The **key** function does for keys what the **id** function does for IDs. The first argument specifies the name of the key. The value of the argument must be a QName, which is expanded as described in [**2.4 Qualified Names**]. When the second argument to the **key** function is of type node-set, then the result is the union of the result of applying the **key** function to the string value of each of the nodes in the argument node-set. When the second argument to **key** is of any other type, the argument is converted to a string as if by a call to the **string** function; it returns a node-set containing the nodes in the same document as the context node that have a value for the named key equal to this string.

For example, given a declaration

```
<xsl:key name="idkey" match="div" use="@id"/>
```

an expression `key("idkey",@ref)` will return the same node-set as `id(@ref)`, assuming that the only ID attribute declared in the XML source document is:

```
<!ATTLIST div id ID #IMPLIED>
```

and that the `ref` attribute of the current node contains no whitespace.

Suppose a document describing a function library uses a `prototype` element to define functions

```
<prototype name="key" return-type="node-set">
<arg type="string"/>
<arg type="object"/>
</prototype>
```

and a `function` element to refer to function names

```
<function>key</function>
```

Then the stylesheet could generate hyperlinks between the references and definitions as follows:

```
<xsl:key name="func" match="prototype" use="@name"/>

<xsl:template match="function">
<b>
  <a href="#{generate-id(key('func',.))}">
    <xsl:apply-templates/>
  </a>
</b>
</xsl:template>

<xsl:template match="prototype">
<p><a name="{generate-id()}">
<b>Function: </b>
...
</a></p>
</xsl:template>
```

The **key** can be used to retrieve a key from a document other than the document containing the context node. For example, suppose a document contains bibliographic references in the form `<bibref>XSLT</bibref>`, and there is a separate XML document `bib.xml` containing a bibliographic database with entries in the form:

```
<entry name="XSLT">...</entry>
```

Then the stylesheet could use the following to transform the `bibref` elements:

```
<xsl:key name="bib" match="entry" use="@name"/>

<xsl:template match="bibref">
  <xsl:variable name="name" select="."/>
  <xsl:for-each select="document('bib.xml')">
    <xsl:apply-templates select="key('bib',$name)"/>
  </xsl:for-each>
</xsl:template>
```

12.3 Number Formatting

Function: *string* **format-number**(*number*, *string*, *string*?)

The **format-number** function converts its first argument to a string using the format pattern string specified by the second argument and the decimal-format named by the third argument, or the default decimal-format, if there is no third argument. The format pattern string is in the syntax specified by the JDK 1.1 DecimalFormat class. The format pattern string is in a localized notation: the decimal-format determines what characters have a special meaning in the pattern (with the exception of the quote character, which is not localized). The format pattern must not contain the currency sign (#x00A4); support for this feature was added after the initial release of JDK 1.1. The decimal-format name must be a QName, which is expanded as described in [**2.4 Qualified Names**]. It is an error if the stylesheet does not contain a declaration of the decimal-format with the specified expanded-name.

NOTE: Implementations are not required to use the JDK 1.1 implementation, nor are implementations required to be implemented in Java.

NOTE: Stylesheets can use other facilities in XPath to control rounding.

```
<!-- Category: top-level-element -->
<xsl:decimal-format
  name = qname
  decimal-separator = char
  grouping-separator = char
  infinity = string
  minus-sign = char
  NaN = string
  percent = char
  per-mille = char
  zero-digit = char
  digit = char
  pattern-separator = char />
```

The `xsl:decimal-format` element declares a decimal-format, which controls the interpretation of a format pattern used by the **format-number** function. If there is a `name` attribute, then the element declares a named decimal-format; otherwise, it declares the default decimal-format. The value of the `name` attribute is a QName, which is expanded as described in [**2.4 Qualified Names**]. It is an error to declare either the default decimal-format or a decimal-format with a given name more than once (even with different import precedence), unless it is declared every time with the same value for all attributes (taking into account any default values).

The other attributes on `xsl:decimal-format` correspond to the methods on the JDK 1.1 DecimalFormatSymbols class. For each `get`/`set` method pair there is an attribute defined for the `xsl:decimal-format` element.

The following attributes both control the interpretation of characters in the format pattern and specify characters that may appear in the result of formatting the number:

- `decimal-separator` specifies the character used for the decimal sign; the default value is the period character (`.`)
- `grouping-separator` specifies the character used as a grouping (e.g. thousands) separator; the default value is the comma character (`,`)
- `percent` specifies the character used as a percent sign; the default value is the percent character (`%`)
- `per-mille` specifies the character used as a per mille sign; the default value is the Unicode per-mille character (#x2030)
- `zero-digit` specifies the character used as the digit zero; the default value is the digit zero (`0`)

The following attributes control the interpretation of characters in the format pattern:

- `digit` specifies the character used for a digit in the format pattern; the default value is the number sign character (`#`)
- `pattern-separator` specifies the character used to separate positive and negative sub patterns in a pattern; the default value is the semi-colon character (`;`)

The following attributes specify characters or strings that may appear in the result of formatting the number:

- `infinity` specifies the string used to represent infinity; the default value is the string `Infinity`
- `NaN` specifies the string used to represent the NaN value; the default value is the string `NaN`
- `minus-sign` specifies the character used as the default minus sign; the default value is the hyphen-minus character (`-`, #x2D)

12.4 Miscellaneous Additional Functions

Function: *node-set* **current**()

The **current** function returns a node-set that has the current node as its only member. For an outermost expression (an expression not occurring within another expression), the current node is always the same as the context node. Thus,

```
<xsl:value-of select="current()"/>
```

means the same as

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

However, within square brackets the current node is usually different from the context node. For example,

```
<xsl:apply-templates select="//glossary/item[@name=current()/@ref]"/>
```

will process all `item` elements that have a `glossary` parent element and that have a `name` attribute with value equal to the value of the current node's `ref` attribute. This is different from

```
<xsl:apply-templates select="//glossary/item[@name=./@ref]"/>
```

which means the same as

```
<xsl:apply-templates select="//glossary/item[@name=@ref]"/>
```

and so would process all `item` elements that have a `glossary` parent element and that have a `name` attribute and a `ref` attribute with the same value.

It is an error to use the **current** function in a pattern.

Function: *string* **unparsed-entity-uri**(*string*)

The **unparsed-entity-uri** returns the URI of the unparsed entity with the specified name in the same document as the context node (see [**3.3 Unparsed Entities**]). It returns the empty string if there is no such entity.

Function: *string* **generate-id**(*node-set*?)

The **generate-id** function returns a string that uniquely identifies the node in the argument node-set that is first in document order. The unique identifier must consist of ASCII alphanumeric characters and must start with an alphabetic character. Thus, the string is syntactically an XML name. An implementation is free to generate an identifier in any convenient way provided that it always generates the same identifier for the same node and that different identifiers are always generated from different nodes. An implementation is under no obligation to generate the same identifiers each time a document is transformed. There is no guarantee that a generated unique identifier will be distinct from any unique IDs specified in the source document. If the argument node-set is empty, the empty string is returned. If the argument is omitted, it defaults to the context node.

Function: *object* **system-property**(*string*)

The argument must evaluate to a string that is a QName. The QName is expanded into a name using the namespace declarations in scope for the expression. The **system-property** function returns an object representing the value of the system property identified by the name. If there is no such system property, the empty string should be returned.

Implementations must provide the following system properties, which are all in the XSLT namespace:

- `xsl:version`, a number giving the version of XSLT implemented by the processor; for XSLT processors implementing the version of XSLT specified by this document, this is the number 1.0
- `xsl:vendor`, a string identifying the vendor of the XSLT processor
- `xsl:vendor-url`, a string containing a URL identifying the vendor of the XSLT processor; typically this is the host page (home page) of the vendor’s Web site.

13 Messages

```
<!-- Category: instruction -->
<xsl:message
  terminate = "yes" | "no">
  <!-- Content: template -->
</xsl:message>
```

The `xsl:message` instruction sends a message in a way that is dependent on the XSLT processor. The content of the `xsl:message` instruction is a template. The `xsl:message` is instantiated by instantiating the content to create an XML fragment. This XML fragment is the content of the message.

> **NOTE:** An XSLT processor might implement `xsl:message` by popping up an alert box or by writing to a log file.

If the `terminate` attribute has the value `yes`, then the XSLT processor should terminate processing after sending the message. The default value is `no`.

One convenient way to do localization is to put the localized information (message text, etc.) in an XML document, which becomes an additional input file to the stylesheet. For example, suppose messages for a language *L* are stored in an XML file `resources/L.xml` in the form:

```
<messages>
  <message name="problem">A problem was detected.</message>
  <message name="error">An error was detected.</message>
</messages>
```

Then a stylesheet could use the following approach to localize messages:

```
<xsl:param name="lang" select="en"/>
<xsl:variable name="messages"
  select="document(concat('resources/', $lang, '.xml'))/messages"/>

<xsl:template name="localized-message">
  <xsl:param name="name"/>
  <xsl:message>
    <xsl:value-of select="$messages/message[@name=$name]"/>
  </xsl:message>
</xsl:template>

<xsl:template name="problem">
  <xsl:call-template name="localized-message"/>
    <xsl:with-param name="name">problem</xsl:with-param>
  </xsl:call-template>
</xsl:template>
```

14 Extensions

XSLT allows two kinds of extension, extension elements and extension functions.

This version of XSLT does not provide a mechanism for defining implementations of extensions. Therefore, an XSLT stylesheet that must be portable between XSLT implementations cannot rely on particular extensions being available. XSLT provides mechanisms that allow an XSLT stylesheet to determine whether the XSLT processor by which it is being processed has implementations of particular extensions available, and to specify what should happen if those extensions are not available. If an XSLT stylesheet is careful to make use of these mechanisms, it is possible for it to take advantage of extensions and still work with any XSLT implementation.

14.1 Extension Elements

The element extension mechanism allows namespaces to be designated as **extension namespace**s. When a namespace is designated as an extension namespace and an element with a name from that namespace occurs in a template, then the element is treated as an instruction rather than as a literal result element. The namespace determines the semantics of the instruction.

> **NOTE:** Since an element that is a child of an `xsl:stylesheet` element is not occurring *in a template*, non-XSLT top-level elements are not extension elements as defined here, and nothing in this section applies to them.

A namespace is designated as an extension namespace by using an `extension-element-prefixes` attribute on an `xsl:stylesheet` element or an `xsl:extension-element-prefixes` attribute on a literal result element. The value of both these attributes is a whitespace-separated list of namespace prefixes. The namespace bound to each of the prefixes is designated as an extension namespace. It is an error if there is no namespace bound to the prefix on the element bearing the `extension-element-prefixes` or `xsl:extension-element-prefixes` attribute. The default namespace (as declared by `xmlns`) may be designated as an extension namespace by including `#default` in the list of namespace prefixes. The designation of a namespace as an extension namespace is effective within the subtree of the stylesheet rooted at the element bearing the `extension-element-prefixes` or `xsl:extension-element-prefixes` attribute.

If the XSLT processor does not have an implementation of a particular extension element available, then the **element-available** function must return false for the name of the element. When such an extension element is instantiated, then the XSLT processor must perform fallback for the element as specified in [**15 Fallback**]. An XSLT processor must not signal an error merely because a template contains an extension element for which no implementation is available.

If the XSLT processor has an implementation of a particular extension element available, then the **element-available** function must return true for the name of the element.

14.2 Extension Functions

If a FunctionName in a FunctionCall expression is not an NCName (i.e. if it contains a colon), then it is treated as a call to an extension function. The FunctionName is expanded to a name using the namespace declarations from the evaluation context.

If the XSLT processor does not have an implementation of an extension function of a particular name available, then the **function-available** function must return false for that name. If such an extension function occurs in an expression and the extension function is actually called, the XSLT processor must signal an error. An XSLT processor must not signal an error merely because an expression contains an extension function for which no implementation is available.

If the XSLT processor has an implementation of an extension function of a particular name available, then the **function-available** function must return true for that name. If such an extension is called, then the XSLT processor must call the implementation passing it the function call arguments; the result returned by the implementation is returned as the result of the function call.

15 Fallback

```
<!-- Category: instruction -->
<xsl:fallback>
  <!-- Content: template -->
</xsl:fallback>
```

Normally, instantiating an `xsl:fallback` element does nothing. However, when an XSLT processor performs fallback for an instruction element, if the instruction element has one or more `xsl:fallback` children, then the content of each of the `xsl:fallback` children must be instantiated in sequence; otherwise, an error must be signaled. The content of an `xsl:fallback` element is a template.

The following functions can be used with the `xsl:choose` and `xsl:if` instructions to explicitly control how a stylesheet should behave if particular elements or functions are not available.

Function: *boolean* **element-available**(*string*)

The argument must evaluate to a string that is a QName. The QName is expanded into an expanded-name using the namespace declarations in scope for the expression. The **element-available** function returns true if and only if the expanded-name is the name of an instruction. If the expanded-name has a namespace URI equal to the XSLT namespace URI, then it refers to an element defined by XSLT. Otherwise, it refers to an extension element. If the expanded-name has a null namespace URI, the **element-available** function will return false.

Function: *boolean* **function-available**(*string*)

The argument must evaluate to a string that is a QName. The QName is expanded into an expanded-name using the namespace declarations in scope for the expression. The **function-available** function returns true if and only if the expanded-name is the name of a function in the function library. If the expanded-name has a non-null namespace URI, then it refers to an extension function; otherwise, it refers to a function defined by XPath or XSLT.

16 Output

```
<!-- Category: top-level-element -->
<xsl:output
  method = "xml" | "html" | "text" | qname-but-not-ncname
  version = nmtoken
  encoding = string
  omit-xml-declaration = "yes" | "no"
  standalone = "yes" | "no"
  doctype-public = string
  doctype-system = string
  cdata-section-elements = qnames
  indent = "yes" | "no"
  media-type = string />
```

An XSLT processor may output the result tree as a sequence of bytes, although it is not required to be able to do so (see [**17 Conformance**]). The `xsl:output` element allows stylesheet authors to specify how they wish the result tree to be output. If an XSLT processor outputs the result tree, it should do so as specified by the `xsl:output` element; however, it is not required to do so.

The `xsl:output` element is only allowed as a top-level element.

The `method` attribute on `xsl:output` identifies the overall method that should be used for outputting the result tree. The value must be a QName. If the QName does not have a prefix, then it identifies a method specified in this document and must be one of `xml`, `html` or `text`. If the QName has a prefix, then the QName is expanded into an expanded-name as described in [**2.4 Qualified Names**]; the expanded-name identifies the output method; the behavior in this case is not specified by this document.

The default for the `method` attribute is chosen as follows. If

- the root node of the result tree has an element child,
- the expanded-name of the first element child of the root node (i.e. the document element) of the result tree has local part `html` (in any combination of upper and lower case) and a null namespace URI, and
- any text nodes preceding the first element child of the root node of the result tree contain only whitespace characters,

then the default output method is `html`; otherwise, the default output method is `xml`. The default output method should be used if there are no `xsl:output` elements or if none of the `xsl:output` elements specifies a value for the `method` attribute.

The other attributes on `xsl:output` provide parameters for the output method. The following attributes are allowed:

- `version` specifies the version of the output method
- `indent` specifies whether the XSLT processor may add additional whitespace when outputting the result tree; the value must be `yes` or `no`
- `encoding` specifies the preferred character encoding that the XSLT processor should use to encode sequences of characters as sequences of bytes; the value of the attribute should be treated case-insensitively; the value must contain only characters in the range #x21 to #x7E (i.e. printable ASCII characters); the value should either be a `charset` registered with the Internet Assigned Numbers Authority [IANA] or start with `X-`
- `media-type` specifies the media type (MIME content type) of the data that results from outputting the result tree; the `charset` parameter should not be specified explicitly; instead, when the top-level media type is `text`, a `charset` parameter should be added according to the character encoding actually used by the output method
- `doctype-system` specifies the system identifier to be used in the document type declaration
- `doctype-public` specifies the public identifier to be used in the document type declaration
- `omit-xml-declaration` specifies whether the XSLT processor should output an XML declaration; the value must be `yes` or `no`
- `standalone` specifies whether the XSLT processor should output a standalone document declaration; the value must be `yes` or `no`
- `cdata-section-elements` specifies a list of the names of elements whose text node children should be output using CDATA sections

The detailed semantics of each attribute will be described separately for each output method for which it is applicable. If the semantics of an attribute are not described for an output method, then it is not applicable to that output method.

A stylesheet may contain multiple `xsl:output` elements and may include or import stylesheets that also contain `xsl:output` elements. All the `xsl:output` elements occurring in a stylesheet are merged into a single effective `xsl:output` element. For the `cdata-section-elements` attribute, the effective value is the union of the specified values. For other attributes, the effective value is the specified value with the highest import precedence. It is an error if there is more than one such value for an attribute. An XSLT processor may signal the error; if it does not signal the error, if should recover by using the value that occurs last in the stylesheet. The values of attributes are defaulted after the `xsl:output` elements have been merged; different output methods may have different default values for an attribute.

16.1 XML Output Method

The `xml` output method outputs the result tree as a well-formed XML external general parsed entity. If the root node of the result tree has a single element node child and no text node children, then the entity should also be a well-formed XML document entity. When the entity is referenced within a trivial XML document wrapper like this

```
<!DOCTYPE doc [
<!ENTITY e SYSTEM "entity-URI">
]>
<doc>&e;</doc>
```

where *`entity-URI`* is a URI for the entity, then the wrapper document as a whole should be a well-formed XML document conforming to the XML Namespaces Recommendation [XML Names]. In addition, the output should be such that if a new tree was constructed by parsing the wrapper as an XML document as specified in [**3 Data Model**], and then removing the document element, making its children instead be children of the root node, then the new tree would be the same as the result tree, with the following possible exceptions:

- The order of attributes in the two trees may be different.
- The new tree may contain namespace nodes that were not present in the result tree.

> **NOTE:** An XSLT processor may need to add namespace declarations in the course of outputting the result tree as XML.

If the XSLT processor generated a document type declaration because of the `doctype-system` attribute, then the above requirements apply to the entity with the generated document type declaration removed.

The `version` attribute specifies the version of XML to be used for outputting the result tree. If the XSLT processor does not support this version of XML, it should use a version of XML that it does support. The version output in the XML declaration (if an XML declaration is output) should correspond to the version of XML that the processor used for outputting the result tree. The value of the `version` attribute should match the VersionNum production of the XML Recommendation [XML]. The default value is `1.0`.

The `encoding` attribute specifies the preferred encoding to use for outputting the result tree. XSLT processors are required to respect values of `UTF-8` and `UTF-16`. For other values, if the XSLT processor does not support the specified encoding it may signal an error; if it does not signal an error it should use

`UTF-8` or `UTF-16` instead. The XSLT processor must not use an encoding whose name does not match the EncName production of the XML Recommendation [XML]. If no `encoding` attribute is specified, then the XSLT processor should use either `UTF-8` or `UTF-16`. It is possible that the result tree will contain a character that cannot be represented in the encoding that the XSLT processor is using for output. In this case, if the character occurs in a context where XML recognizes character references (i.e. in the value of an attribute node or text node), then the character should be output as a character reference; otherwise (for example if the character occurs in the name of an element) the XSLT processor should signal an error.

If the `indent` attribute has the value `yes`, then the `xml` output method may output whitespace in addition to the whitespace in the result tree (possibly based on whitespace stripped from either the source document or the stylesheet) in order to indent the result nicely; if the `indent` attribute has the value `no`, it should not output any additional whitespace. The default value is `no`. The `xml` output method should use an algorithm to output additional whitespace that ensures that the result if whitespace were to be stripped from the output using the process described in [**3.4 Whitespace Stripping**] with the set of whitespace-preserving elements consisting of just `xsl:text` would be the same when additional whitespace is output as when additional whitespace is not output.

> **NOTE:** It is usually not safe to use `indent="yes"` with document types that include element types with mixed content.

The `cdata-section-elements` attribute contains a whitespace-separated list of QNames. Each QName is expanded into an expanded-name using the namespace declarations in effect on the `xsl:output` element in which the QName occurs; if there is a default namespace, it is used for QNames that do not have a prefix. The expansion is performed before the merging of multiple `xsl:output` elements into a single effective `xsl:output` element. If the expanded-name of the parent of a text node is a member of the list, then the text node should be output as a CDATA section. For example,

```
<xsl:output cdata-section-elements="example"/>
```

would cause a literal result element written in the stylesheet as

```
<example>&lt;foo></example>
```

or as

```
<example><![CDATA[<foo>]]></example>
```

to be output as

```
<example><![CDATA[<foo>]]></example>
```

If the text node contains the sequence of characters `]]>`, then the currently open CDATA section should be closed following the `]]` and a new CDATA section opened before the `>`. For example, a literal result element written in the stylesheet as

```
<example>]]&gt;</example>
```

would be output as

```
<example><![CDATA[]]]]><![CDATA[>]]></example>
```

If the text node contains a character that is not representable in the character encoding being used to output the result tree, then the currently open CDATA section should be closed before the character, the character should be output using a character reference or entity reference, and a new CDATA section should be opened for any further characters in the text node.

CDATA sections should not be used except for text nodes that the `cdata-section-elements` attribute explicitly specifies should be output using CDATA sections.

The `xml` output method should output an XML declaration unless the `omit-xml-declaration` attribute has the value `yes`. The XML declaration should include both version information and an encoding declaration. If the `standalone` attribute is specified, it should include a standalone document declaration with the same value as the value as the value of the `standalone` attribute. Otherwise, it should not include a standalone document declaration; this ensures that it is both a XML declaration (allowed at the beginning of a document entity) and a text declaration (allowed at the beginning of an external general parsed entity).

If the `doctype-system` attribute is specified, the `xml` output method should output a document type declaration immediately before the first element. The name following `<!DOCTYPE` should be the name of the first element. If `doctype-public` attribute is also specified, then the `xml` output method should output `PUBLIC` followed by the public identifier and then the system identifier; otherwise, it should output `SYSTEM` followed by the system identifier. The internal subset should be empty. The `doctype-public` attribute should be ignored unless the `doctype-system` attribute is specified.

The `media-type` attribute is applicable for the `xml` output method. The default value for the `media-type` attribute is `text/xml`.

16.2 HTML Output Method

The `html` output method outputs the result tree as HTML; for example,

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

<xsl:output method="html"/>

<xsl:template match="/">
  <html>
   <xsl:apply-templates/>
  </html>
</xsl:template>

...

</xsl:stylesheet>
```

The `version` attribute indicates the version of the HTML. The default value is `4.0`, which specifies that the result should be output as HTML conforming to the HTML 4.0 Recommendation.

The `html` output method should not output an element differently from the `xml` output method unless the expanded-name of the element has a null namespace URI; an element whose expanded-name has a non-null namespace URI should be output as XML. If the expanded-name of the element has a null namespace URI, but the local part of the expanded-name is not recognized as the name of an HTML element, the element should output in the same way as a non-empty, inline element such as `span`.

The `html` output method should not output an end-tag for empty elements. For HTML 4.0, the empty elements are `area`, `base`, `basefont`, `br`, `col`, `frame`, `hr`, `img`, `input`, `isindex`, `link`, `meta` and `param`. For example, an element written as
 or
</br> in the stylesheet should be output as
.

The `html` output method should recognize the names of HTML elements regardless of case. For example, elements named `br`, `BR` or `Br` should all be recognized as the HTML `br` element and output without an end-tag.

The `html` output method should not perform escaping for the content of the `script` and `style` elements. For example, a literal result element written in the stylesheet as

```
<script>if (a &lt; b) foo()</script>
```

or

```
<script><![CDATA[if (a < b) foo()]]></script>
```

should be output as

```
<script>if (a < b) foo()</script>
```

The `html` output method should not escape < characters occurring in attribute values.

If the `indent` attribute has the value `yes`, then the `html` output method may add or remove whitespace as it outputs the result tree, so long as it does not change how an HTML user agent would render the output. The default value is `yes`.

The `html` output method should escape non-ASCII characters in URI attribute values using the method recommended in Section B.2.1 of the HTML 4.0 Recommendation.

The `html` output method may output a character using a character entity reference, if one is defined for it in the version of HTML that the output method is using.

The `html` output method should terminate processing instructions with > rather than `?>`.

The `html` output method should output boolean attributes (that is attributes with only a single allowed value that is equal to the name of the attribute) in minimized form. For example, a start-tag written in the stylesheet as

```
<OPTION selected="selected">
```

should be output as

```
<OPTION selected>
```

The `html` output method should not escape a `&` character occurring in an attribute value immediately followed by a `{` character (see Section B.7.1 of the HTML 4.0 Recommendation). For example, a start-tag written in the stylesheet as

```
<BODY bgcolor='&amp;{{randomrbg}};'>
```

should be output as

```
<BODY bgcolor='&{randomrbg};'>
```

The `encoding` attribute specifies the preferred encoding to be used. If there is a `HEAD` element, then the `html` output method should add a `META` element immediately after the start-tag of the `HEAD` element specifying the character encoding actually used. For example,

```
<HEAD>
<META http-equiv="Content-Type" content="text/html; charset=EUC-JP">
...
```

It is possible that the result tree will contain a character that cannot be represented in the encoding that the XSLT processor is using for output. In this case, if the character occurs in a context where HTML recognizes character references, then the character should be output as a character reference; otherwise (for example, in a `script` or `style` element or in a comment), the XSLT processor should signal an error.

If the `doctype-public` or `doctype-system` attributes are specified, then the `html` output method should output a document type declaration immediately before the first element. The name following `<!DOCTYPE` should be `HTML` or `html`. If the `doctype-public` attribute is specified, then the output method should output `PUBLIC` followed by the specified public identifier; if the `doctype-system` attribute is also specified, it should also output the specified system identifier following the public identifier. If the `doctype-system` attribute is specified but the `doctype-public` attribute is not specified, then the output method should output `SYSTEM` followed by the specified system identifier.

The `media-type` attribute is applicable for the `html` output method. The default value is `text/html`.

16.3 Text Output Method

The `text` output method outputs the result tree by outputting the string-value of every text node in the result tree in document order without any escaping.

The `media-type` attribute is applicable for the `text` output method. The default value for the `media-type` attribute is `text/plain`.

The `encoding` attribute identifies the encoding that the `text` output method should use to convert sequences of characters to sequences of bytes. The default is system-dependent. If the result tree contains a character that cannot be represented in the encoding that the XSLT processor is using for output, the XSLT processor should signal an error.

16.4 Disabling Output Escaping

Normally, the `xml` output method escapes & and < (and possibly other characters) when outputting text nodes. This ensures that the output is well-formed XML. However, it is sometimes convenient to be able to produce output that is almost, but not quite well-formed XML; for example, the output may include ill-formed sections which are intended to be transformed into well-formed XML by a subsequent non-XML aware process. For this reason, XSLT provides a mechanism for disabling output escaping. An `xsl:value-of` or `xsl:text` element may have a `disable-output-escaping` attribute; the allowed values are `yes` or `no`; the default is `no`; if the value is `yes`, then a text node generated by instantiating the `xsl:value-of` or `xsl:text` element should be output without any escaping. For example,

```
<xsl:text disable-output-escaping="yes">&lt;</xsl:text>
```

should generate the single character `<`.

It is an error for output escaping to be disabled for a text node that is used for something other than a text node in the result tree. Thus, it is an error to disable output escaping for an `xsl:value-of` or `xsl:text` element that is used to generate the string-value of a comment, processing instruction or attribute node; it is also an error to convert a result tree fragment to a number or a string if the result tree fragment contains a text node for which escaping was disabled. In both cases, an XSLT processor may signal the error; if it does not signal the error, it must recover by ignoring the `disable-output-escaping` attribute.

The `disable-output-escaping` attribute may be used with the `html` output method as well as with the `xml` output method. The `text` output method ignores the `disable-output-escaping` attribute, since it does not perform any output escaping.

An XSLT processor will only be able to disable output escaping if it controls how the result tree is output. This may not always be the case. For example, the result tree may be used as the source tree for another XSLT transformation instead of being output. An XSLT processor is not required to support disabling output escaping. If an `xsl:value-of` or `xsl:text` specifies that output escaping should be disabled and the XSLT processor does not support this, the XSLT processor may signal an error; if it does not signal an error, it must recover by not disabling output escaping.

If output escaping is disabled for a character that is not representable in the encoding that the XSLT processor is using for output, then the XSLT processor may signal an error; if it does not signal an error, it must recover by not disabling output escaping.

Since disabling output escaping may not work with all XSLT processors and can result in XML that is not well-formed, it should be used only when there is no alternative.

17 Conformance

A conforming XSLT processor must be able to use a stylesheet to transform a source tree into a result tree as specified in this document. A conforming XSLT processor need not be able to output the result in XML or in any other form.

> **NOTE:** Vendors of XSLT processors are strongly encouraged to provide a way to verify that their processor is behaving conformingly by allowing the result tree to be output as XML or by providing access to the result tree through a standard API such as the DOM or SAX.

A conforming XSLT processor must signal any errors except for those that this document specifically allows an XSLT processor not to signal. A conforming XSLT processor may but need not recover from any errors that it signals.

A conforming XSLT processor may impose limits on the processing resources consumed by the processing of a stylesheet.

18 Notation

The specification of each XSLT-defined element type is preceded by a summary of its syntax in the form of a model for elements of that element type. The meaning of syntax summary notation is as follows:

- An attribute is required if and only if its name is in bold.
- The string that occurs in the place of an attribute value specifies the allowed values of the attribute. If this is surrounded by curly braces, then the attribute value is treated as an attribute value template, and the string occurring within curly braces specifies the allowed values of the result of instantiating the attribute value template. Alternative allowed values are separated by `|`. A quoted string indicates a value equal to that specific string. An unquoted, italicized name specifies a particular type of value.
- If the element is allowed not to be empty, then the element contains a comment specifying the allowed content. The allowed content is specified in a similar way to an element type declaration in XML; *template* means that any mixture of text nodes, literal result elements, extension elements, and XSLT elements from the `instruction` category is allowed; *top-level-elements* means that any mixture of XSLT elements from the `top-level-element` category is allowed.
- The element is prefaced by comments indicating if it belongs to the `instruction` category or `top-level-element` category or both. The category of an element just affects whether it is allowed in the content of elements that allow a *template* or *top-level-elements*.

A References

A.1 Normative References

XML
: World Wide Web Consortium. *Extensible Markup Language (XML) 1.0.* W3C Recommendation. See http://www.w3.org/TR/1998/REC-xml-19980210

XML Names
: World Wide Web Consortium. *Namespaces in XML.* W3C Recommendation. See http://www.w3.org/TR/REC-xml-names

XPath
: World Wide Web Consortium. *XML Path Language.* W3C Proposed Recommendation. See http://www.w3.org/TR/xpath

A.2 Other References

CSS2
: World Wide Web Consortium. *Cascading Style Sheets, level 2 (CSS2)*. W3C Recommendation. See http://www.w3.org/TR/1998/REC-CSS2-19980512

DSSSL
: International Organization for Standardization, International Electrotechnical Commission. *ISO/IEC 10179:1996. Document Style Semantics and Specification Language (DSSSL)*. International Standard.

IANA
: Internet Assigned Numbers Authority. *Character Sets*. See ftp://ftp.isi.edu/in-notes/iana/assignments/character-sets.

RFC2376
: E. Whitehead, M. Murata. *XML Media Types*. IETF RFC 2376. See http://www.ietf.org/rfc/rfc2376.txt.

RFC2396
: T. Berners-Lee, R. Fielding, and L. Masinter. *Uniform Resource Identifiers (URI): Generic Syntax*. IETF RFC 2396. See http://www.ietf.org/rfc/rfc2396.txt.

UNICODE TR10
: Unicode Consortium. *Unicode Technical Report #10. Unicode Collation Algorithm*. Unicode Technical Report. See http://www.unicode.org/unicode/reports/tr10/index.html.

XHTML
: World Wide Web Consortium. *XHTML 1.0: The Extensible HyperText Markup Language.* W3C Proposed Recommendation. See http://www.w3.org/TR/xhtml1

XPointer
: World Wide Web Consortium. *XML Pointer Language (XPointer).* W3C Working Draft. See http://www.w3.org/TR/xptr

XML Stylesheet
: World Wide Web Consortium. *Associating stylesheets with XML documents.* W3C Recommendation. See http://www.w3.org/TR/xml-stylesheet

XSL
: World Wide Web Consortium. *Extensible Stylesheet Language (XSL).* W3C Working Draft. See http://www.w3.org/TR/WD-xsl

B Element Syntax Summary

```
<!-- Category: instruction -->
<xsl:apply-imports />

<!-- Category: instruction -->
<xsl:apply-templates
  select = node-set-expression
  mode = qname>
  <!-- Content: (xsl:sort | xsl:with-param)* -->
</xsl:apply-templates>

<!-- Category: instruction -->
<xsl:attribute
  name = { qname }
  namespace = { uri-reference }>
```

```
  <!-- Content: template -->
</xsl:attribute>

<!-- Category: top-level-element -->
<xsl:attribute-set
  name = qname
  use-attribute-sets = qnames>
  <!-- Content: xsl:attribute* -->
</xsl:attribute-set>

<!-- Category: instruction -->
<xsl:call-template
  name = qname>
  <!-- Content: xsl:with-param* -->
</xsl:call-template>

<!-- Category: instruction -->
<xsl:choose>
  <!-- Content: (xsl:when+, xsl:otherwise?) -->
</xsl:choose>

<!-- Category: instruction -->
<xsl:comment>
  <!-- Content: template -->
</xsl:comment>

<!-- Category: instruction -->
<xsl:copy
  use-attribute-sets = qnames>
  <!-- Content: template -->
</xsl:copy>

<!-- Category: instruction -->
<xsl:copy-of
  select = expression />

<!-- Category: top-level-element -->
<xsl:decimal-format
  name = qname
  decimal-separator = char
  grouping-separator = char
  infinity = string
  minus-sign = char
  NaN = string
  percent = char
  per-mille = char
  zero-digit = char
  digit = char
  pattern-separator = char />

<!-- Category: instruction -->
<xsl:element
  name = { qname }
  namespace = { uri-reference }
  use-attribute-sets = qnames>
  <!-- Content: template -->
</xsl:element>
```

```
<!-- Category: instruction -->
<xsl:fallback>
  <!-- Content: template -->
</xsl:fallback>

<!-- Category: instruction -->
<xsl:for-each
  select = node-set-expression>
  <!-- Content: (xsl:sort*, template) -->
</xsl:for-each>

<!-- Category: instruction -->
<xsl:if
  test = boolean-expression>
  <!-- Content: template -->
</xsl:if>

<xsl:import
  href = uri-reference />

<!-- Category: top-level-element -->
<xsl:include
  href = uri-reference />

<!-- Category: top-level-element -->
<xsl:key
  name = qname
  match = pattern
  use = node-set-expression />

<!-- Category: instruction -->
<xsl:message
  terminate = "yes" | "no">
  <!-- Content: template -->
</xsl:message>

<!-- Category: top-level-element -->
<xsl:namespace-alias
  stylesheet-prefix = prefix | "#default"
  result-prefix = prefix | "#default" />

<!-- Category: instruction -->
<xsl:number
  level = "single" | "multiple" | "any"
  count = pattern
  from = pattern
  value = number-expression
  format = { string }
  lang = { nmtoken }
  letter-value = { "alphabetic" | "traditional" }
  grouping-separator = { char }
  grouping-size = { number } />

<xsl:otherwise>
  <!-- Content: template -->
```

```
</xsl:otherwise>

<!-- Category: top-level-element -->
<xsl:output
  method = "xml" | "html" | "text" | qname-but-not-ncname
  version = nmtoken
  encoding = string
  omit-xml-declaration = "yes" | "no"
  standalone = "yes" | "no"
  doctype-public = string
  doctype-system = string
  cdata-section-elements = qnames
  indent = "yes" | "no"
  media-type = string />

<!-- Category: top-level-element -->
<xsl:param
  name = qname
  select = expression>
  <!-- Content: template -->
</xsl:param>

<!-- Category: top-level-element -->
<xsl:preserve-space
  elements = tokens />

<!-- Category: instruction -->
<xsl:processing-instruction
  name = { ncname }>
  <!-- Content: template -->
</xsl:processing-instruction>

<xsl:sort
  select = string-expression
  lang = { nmtoken }
  data-type = { "text" | "number" }
  order = { "ascending" | "descending" }
  case-order = { "upper-first" | "lower-first" } />

<!-- Category: top-level-element -->
<xsl:strip-space
  elements = tokens />

<xsl:stylesheet
  id = id
  extension-element-prefixes = tokens
  exclude-result-prefixes = tokens
  version = number>
  <!-- Content: (xsl:import*, top-level-elements) -->
</xsl:stylesheet>

<!-- Category: top-level-element -->
<xsl:template
  match = pattern
  name = qname
  priority = number
  mode = qname>
```

```
  <!-- Content: (xsl:param*, template) -->
</xsl:template>

<!-- Category: instruction -->
<xsl:text
  disable-output-escaping = "yes" | "no">
  <!-- Content: #PCDATA -->
</xsl:text>

<xsl:transform
  id = id
  extension-element-prefixes = tokens
  exclude-result-prefixes = tokens
  version = number>
  <!-- Content: (xsl:import*, top-level-elements) -->
</xsl:transform>

<!-- Category: instruction -->
<xsl:value-of
  select = string-expression
  disable-output-escaping = "yes" | "no" />

<!-- Category: top-level-element -->
<!-- Category: instruction -->
<xsl:variable
  name = qname
  select = expression>
  <!-- Content: template -->
</xsl:variable>

<xsl:when
  test = boolean-expression>
  <!-- Content: template -->
</xsl:when>

<xsl:with-param
  name = qname
  select = expression>
  <!-- Content: template -->
</xsl:with-param>
```

C DTD Fragment for XSLT Stylesheets (Non-Normative)

> **NOTE:** This DTD Fragment is not normative because XML 1.0 DTDs do not support XML Namespaces and thus cannot correctly describe the allowed structure of an XSLT stylesheet.

The following entity can be used to construct a DTD for XSLT stylesheets that create instances of a particular result DTD. Before referencing the entity, the stylesheet DTD must define a `result-elements` parameter entity listing the allowed result element types. For example:

```
<!ENTITY % result-elements "
  | fo:inline-sequence
  | fo:block
">
```

Such result elements should be declared to have `xsl:use-attribute-sets` and `xsl:extension-element-prefixes` attributes. The following entity declares the `result-element-atts` parameter for this purpose. The content that XSLT allows for result elements is the same as it allows for the XSLT elements that are declared in the following entity with a content model of `%template;`. The DTD may use a more restrictive content model than `%template;` to reflect the constraints of the result DTD.

The DTD may define the `non-xsl-top-level` parameter entity to allow additional top-level elements from namespaces other than the XSLT namespace.

The use of the `xsl:` prefix in this DTD does not imply that XSLT stylesheets are required to use this prefix. Any of the elements declared in this DTD may have attributes whose name starts with `xmlns:` or is equal to `xmlns` in addition to the attributes declared in this DTD.

```
<!ENTITY % char-instructions "
  | xsl:apply-templates
  | xsl:call-template
  | xsl:apply-imports
  | xsl:for-each
  | xsl:value-of
  | xsl:copy-of
  | xsl:number
  | xsl:choose
  | xsl:if
  | xsl:text
  | xsl:copy
  | xsl:variable
  | xsl:message
  | xsl:fallback
">

<!ENTITY % instructions "
  %char-instructions;
  | xsl:processing-instruction
  | xsl:comment
  | xsl:element
  | xsl:attribute
">

<!ENTITY % char-template "
 (#PCDATA
  %char-instructions;)*
">

<!ENTITY % template "
 (#PCDATA
  %instructions;
  %result-elements;)*
">

<!-- Used for the type of an attribute value that is a URI reference.-->
<!ENTITY % URI "CDATA">

<!-- Used for the type of an attribute value that is a pattern.-->
<!ENTITY % pattern "CDATA">

<!-- Used for the type of an attribute value that is an
```

```
     attribute value template.-->
<!ENTITY % avt "CDATA">

<!-- Used for the type of an attribute value that is a QName; the prefix
     gets expanded by the XSLT processor. -->
<!ENTITY % qname "NMTOKEN">

<!-- Like qname but a whitespace-separated list of QNames. -->
<!ENTITY % qnames "NMTOKENS">

<!-- Used for the type of an attribute value that is an expression.-->
<!ENTITY % expr "CDATA">

<!-- Used for the type of an attribute value that consists
     of a single character.-->
<!ENTITY % char "CDATA">

<!-- Used for the type of an attribute value that is a priority. -->
<!ENTITY % priority "NMTOKEN">

<!ENTITY % space-att "xml:space (default|preserve) #IMPLIED">

<!-- This may be overridden to customize the set of elements allowed
at the top-level. -->

<!ENTITY % non-xsl-top-level "">

<!ENTITY % top-level "
 (xsl:import*,
  (xsl:include
  | xsl:strip-space
  | xsl:preserve-space
  | xsl:output
  | xsl:key
  | xsl:decimal-format
  | xsl:attribute-set
  | xsl:variable
  | xsl:param
  | xsl:template
  | xsl:namespace-alias
  %non-xsl-top-level;)*)
">

<!ENTITY % top-level-atts '
  extension-element-prefixes CDATA #IMPLIED
  exclude-result-prefixes CDATA #IMPLIED
  id ID #IMPLIED
  version NMTOKEN #REQUIRED
  xmlns:xsl CDATA #FIXED "http://www.w3.org/1999/XSL/Transform"
  %space-att;
'>

<!-- This entity is defined for use in the ATTLIST declaration
for result elements. -->

<!ENTITY % result-element-atts '
  xsl:extension-element-prefixes CDATA #IMPLIED
  xsl:exclude-result-prefixes CDATA #IMPLIED
  xsl:use-attribute-sets %qnames; #IMPLIED
  xsl:version NMTOKEN #IMPLIED
'>
```

```
<!ELEMENT xsl:stylesheet %top-level;>
<!ATTLIST xsl:stylesheet %top-level-atts;>

<!ELEMENT xsl:transform %top-level;>
<!ATTLIST xsl:transform %top-level-atts;>

<!ELEMENT xsl:import EMPTY>
<!ATTLIST xsl:import href %URI; #REQUIRED>

<!ELEMENT xsl:include EMPTY>
<!ATTLIST xsl:include href %URI; #REQUIRED>

<!ELEMENT xsl:strip-space EMPTY>
<!ATTLIST xsl:strip-space elements CDATA #REQUIRED>

<!ELEMENT xsl:preserve-space EMPTY>
<!ATTLIST xsl:preserve-space elements CDATA #REQUIRED>

<!ELEMENT xsl:output EMPTY>
<!ATTLIST xsl:output
  method %qname; #IMPLIED
  version NMTOKEN #IMPLIED
  encoding CDATA #IMPLIED
  omit-xml-declaration (yes|no) #IMPLIED
  standalone (yes|no) #IMPLIED
  doctype-public CDATA #IMPLIED
  doctype-system CDATA #IMPLIED
  cdata-section-elements %qnames; #IMPLIED
  indent (yes|no) #IMPLIED
  media-type CDATA #IMPLIED
>

<!ELEMENT xsl:key EMPTY>
<!ATTLIST xsl:key
  name %qname; #REQUIRED
  match %pattern; #REQUIRED
  use %expr; #REQUIRED
>

<!ELEMENT xsl:decimal-format EMPTY>
<!ATTLIST xsl:decimal-format
  name %qname; #IMPLIED
  decimal-separator %char; "."
  grouping-separator %char; ","
  infinity CDATA "Infinity"
  minus-sign %char; "-"
  NaN CDATA "NaN"
  percent %char; "%"
  per-mille %char; "&#x2030;"
  zero-digit %char; "0"
  digit %char; "#"
  pattern-separator %char; ";"
>

<!ELEMENT xsl:namespace-alias EMPTY>
<!ATTLIST xsl:namespace-alias
  stylesheet-prefix CDATA #REQUIRED
  result-prefix CDATA #REQUIRED
>

<!ELEMENT xsl:template
```

```
 (#PCDATA
  %instructions;
  %result-elements;
  | xsl:param)*
>

<!ATTLIST xsl:template
  match %pattern; #IMPLIED
  name %qname; #IMPLIED
  priority %priority; #IMPLIED
  mode %qname; #IMPLIED
  %space-att;
>

<!ELEMENT xsl:value-of EMPTY>
<!ATTLIST xsl:value-of
  select %expr; #REQUIRED
  disable-output-escaping (yes|no) "no"
>

<!ELEMENT xsl:copy-of EMPTY>
<!ATTLIST xsl:copy-of select %expr; #REQUIRED>

<!ELEMENT xsl:number EMPTY>
<!ATTLIST xsl:number
   level (single|multiple|any) "single"
   count %pattern; #IMPLIED
   from %pattern; #IMPLIED
   value %expr; #IMPLIED
   format %avt; '1'
   lang %avt; #IMPLIED
   letter-value %avt; #IMPLIED
   grouping-separator %avt; #IMPLIED
   grouping-size %avt; #IMPLIED
>

<!ELEMENT xsl:apply-templates (xsl:sort|xsl:with-param)*>
<!ATTLIST xsl:apply-templates
  select %expr; "node()"
  mode %qname; #IMPLIED
>

<!ELEMENT xsl:apply-imports EMPTY>

<!-- xsl:sort cannot occur after any other elements or
any non-whitespace character -->

<!ELEMENT xsl:for-each
 (#PCDATA
  %instructions;
  %result-elements;
  | xsl:sort)*
>

<!ATTLIST xsl:for-each
  select %expr; #REQUIRED
  %space-att;
>

<!ELEMENT xsl:sort EMPTY>
<!ATTLIST xsl:sort
```

```
  select %expr; "."
  lang %avt; #IMPLIED
  data-type %avt; "text"
  order %avt; "ascending"
  case-order %avt; #IMPLIED
>

<!ELEMENT xsl:if %template;>
<!ATTLIST xsl:if
  test %expr; #REQUIRED
  %space-att;
>

<!ELEMENT xsl:choose (xsl:when+, xsl:otherwise?)>
<!ATTLIST xsl:choose %space-att;>

<!ELEMENT xsl:when %template;>
<!ATTLIST xsl:when
  test %expr; #REQUIRED
  %space-att;
>

<!ELEMENT xsl:otherwise %template;>
<!ATTLIST xsl:otherwise %space-att;>

<!ELEMENT xsl:attribute-set (xsl:attribute)*>
<!ATTLIST xsl:attribute-set
  name %qname; #REQUIRED
  use-attribute-sets %qnames; #IMPLIED
>

<!ELEMENT xsl:call-template (xsl:with-param)*>
<!ATTLIST xsl:call-template
  name %qname; #REQUIRED
>

<!ELEMENT xsl:with-param %template;>
<!ATTLIST xsl:with-param
  name %qname; #REQUIRED
  select %expr; #IMPLIED
>

<!ELEMENT xsl:variable %template;>
<!ATTLIST xsl:variable
  name %qname; #REQUIRED
  select %expr; #IMPLIED
>

<!ELEMENT xsl:param %template;>
<!ATTLIST xsl:param
  name %qname; #REQUIRED
  select %expr; #IMPLIED
>

<!ELEMENT xsl:text (#PCDATA)>
<!ATTLIST xsl:text
  disable-output-escaping (yes|no) "no"
>

<!ELEMENT xsl:processing-instruction %char-template;>
<!ATTLIST xsl:processing-instruction
```

```
  name %avt; #REQUIRED
  %space-att;
>

<!ELEMENT xsl:element %template;>
<!ATTLIST xsl:element
  name %avt; #REQUIRED
  namespace %avt; #IMPLIED
  use-attribute-sets %qnames; #IMPLIED
  %space-att;
>

<!ELEMENT xsl:attribute %char-template;>
<!ATTLIST xsl:attribute
  name %avt; #REQUIRED
  namespace %avt; #IMPLIED
  %space-att;
>

<!ELEMENT xsl:comment %char-template;>
<!ATTLIST xsl:comment %space-att;>

<!ELEMENT xsl:copy %template;>
<!ATTLIST xsl:copy
  %space-att;
  use-attribute-sets %qnames; #IMPLIED
>

<!ELEMENT xsl:message %template;>
<!ATTLIST xsl:message
  %space-att;
  terminate (yes|no) "no"
>

<!ELEMENT xsl:fallback %template;>
<!ATTLIST xsl:fallback %space-att;>
```

D Examples (Non-Normative)

D.1 Document Example

This example is a stylesheet for transforming documents that conform to a simple DTD into XHTML [XHTML]. The DTD is:

```
<!ELEMENT doc (title, chapter*)>
<!ELEMENT chapter (title, (para|note)*, section*)>
<!ELEMENT section (title, (para|note)*)>
<!ELEMENT title (#PCDATA|emph)*>
<!ELEMENT para (#PCDATA|emph)*>
<!ELEMENT note (#PCDATA|emph)*>
<!ELEMENT emph (#PCDATA|emph)*>
```

The stylesheet is:

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns="http://www.w3.org/TR/xhtml1/strict">
```

```
<xsl:strip-space elements="doc chapter section"/>
<xsl:output
   method="xml"
   indent="yes"
   encoding="iso-8859-1"
/>

<xsl:template match="doc">
 <html>
   <head>
     <title>
       <xsl:value-of select="title"/>
     </title>
   </head>
   <body>
     <xsl:apply-templates/>
   </body>
 </html>
</xsl:template>

<xsl:template match="doc/title">
  <h1>
    <xsl:apply-templates/>
  </h1>
</xsl:template>

<xsl:template match="chapter/title">
  <h2>
    <xsl:apply-templates/>
  </h2>
</xsl:template>

<xsl:template match="section/title">
  <h3>
    <xsl:apply-templates/>
  </h3>
</xsl:template>

<xsl:template match="para">
  <p>
    <xsl:apply-templates/>
  </p>
</xsl:template>

<xsl:template match="note">
  <p class="note">
    <b>NOTE: </b>
    <xsl:apply-templates/>
  </p>
</xsl:template>

<xsl:template match="emph">
  <em>
    <xsl:apply-templates/>
  </em>
</xsl:template>

</xsl:stylesheet>
```

With the following input document

```
<!DOCTYPE doc SYSTEM "doc.dtd">
<doc>
<title>Document Title</title>
<chapter>
<title>Chapter Title</title>
<section>
<title>Section Title</title>
<para>This is a test.</para>
<note>This is a note.</note>
</section>
<section>
<title>Another Section Title</title>
<para>This is <emph>another</emph> test.</para>
<note>This is another note.</note>
</section>
</chapter>
</doc>
```

it would produce the following result

```
<?xml version="1.0" encoding="iso-8859-1"?>
<html xmlns="http://www.w3.org/TR/xhtml1/strict">
<head>
<title>Document Title</title>
</head>
<body>
<h1>Document Title</h1>
<h2>Chapter Title</h2>
<h3>Section Title</h3>
<p>This is a test.</p>
<p class="note">
<b>NOTE: </b>This is a note.</p>
<h3>Another Section Title</h3>
<p>This is <em>another</em> test.</p>
<p class="note">
<b>NOTE: </b>This is another note.</p>
</body>
</html>
```

D.2 Data Example

This is an example of transforming some data represented in XML using three different XSLT stylesheets to produce three different representations of the data, HTML, SVG and VRML.

The input data is:

```
<sales>

        <division id="North">
                <revenue>10</revenue>
                <growth>9</growth>
                <bonus>7</bonus>
        </division>

        <division id="South">
                <revenue>4</revenue>
                <growth>3</growth>
                <bonus>4</bonus>
        </division>
```

```
        <division id="West">
                <revenue>6</revenue>
                <growth>-1.5</growth>
                <bonus>2</bonus>
        </division>

</sales>
```

The following stylesheet, which uses the simplified syntax described in [**2.3 Literal Result Element as Stylesheet**], transforms the data into HTML:

```
<html xsl:version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      lang="en">
    <head>
        <title>Sales Results By Division</title>
    </head>
    <body>
        <table border="1">
            <tr>
                <th>Division</th>
                <th>Revenue</th>
                <th>Growth</th>
                <th>Bonus</th>
            </tr>
            <xsl:for-each select="sales/division">
                <!-- order the result by revenue -->
                <xsl:sort select="revenue"
                          data-type="number"
                          order="descending"/>
                <tr>
                    <td>
                        <em><xsl:value-of select="@id"/></em>
                    </td>
                    <td>
                        <xsl:value-of select="revenue"/>
                    </td>
                    <td>
                        <!-- highlight negative growth in red -->
                        <xsl:if test="growth &lt; 0">
                             <xsl:attribute name="style">
                                 <xsl:text>color:red</xsl:text>
                             </xsl:attribute>
                        </xsl:if>
                        <xsl:value-of select="growth"/>
                    </td>
                    <td>
                        <xsl:value-of select="bonus"/>
                    </td>
                </tr>
            </xsl:for-each>
        </table>
    </body>
</html>
```

The HTML output is:

```
<html lang="en">
<head>
```

```
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<title>Sales Results By Division</title>
</head>
<body>
<table border="1">
<tr>
<th>Division</th><th>Revenue</th><th>Growth</th><th>Bonus</th>
</tr>
<tr>
<td><em>North</em></td><td>10</td><td>9</td><td>7</td>
</tr>
<tr>
<td><em>West</em></td><td>6</td><td style="color:red">-1.5</td><td>2</td>
</tr>
<tr>
<td><em>South</em></td><td>4</td><td>3</td><td>4</td>
</tr>
</table>
</body>
</html>
```

The following stylesheet transforms the data into SVG:

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns="http://www.w3.org/Graphics/SVG/SVG-19990812.dtd">

<xsl:output method="xml" indent="yes" media-type="image/svg"/>

<xsl:template match="/">

<svg width = "3in" height="3in">
    <g style = "stroke: #000000">
        <!-- draw the axes -->
        <line x1="0" x2="150" y1="150" y2="150"/>
        <line x1="0" x2="0" y1="0" y2="150"/>
        <text x="0" y="10">Revenue</text>
        <text x="150" y="165">Division</text>
        <xsl:for-each select="sales/division">
            <!-- define some useful variables -->

            <!-- the bar’s x position -->
            <xsl:variable name="pos"
                          select="(position()*40)-30"/>

            <!-- the bar’s height -->
            <xsl:variable name="height"
                          select="revenue*10"/>

            <!-- the rectangle -->
            <rect x="{$pos}" y="{150-$height}"
                  width="20" height="{$height}"/>

            <!-- the text label -->
            <text x="{$pos}" y="165">
                <xsl:value-of select="@id"/>
            </text>

            <!-- the bar value -->
            <text x="{$pos}" y="{145-$height}">
                <xsl:value-of select="revenue"/>
```

```
            </text>
        </xsl:for-each>
    </g>
</svg>

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

The SVG output is:

```
<svg width="3in" height="3in"
     xmlns="http://www.w3.org/Graphics/SVG/svg-19990412.dtd">
    <g style="stroke: #000000">
        <line x1="0" x2="150" y1="150" y2="150"/>
        <line x1="0" x2="0" y1="0" y2="150"/>
        <text x="0" y="10">Revenue</text>
        <text x="150" y="165">Division</text>
        <rect x="10" y="50" width="20" height="100"/>
        <text x="10" y="165">North</text>
        <text x="10" y="45">10</text>
        <rect x="50" y="110" width="20" height="40"/>
        <text x="50" y="165">South</text>
        <text x="50" y="105">4</text>
        <rect x="90" y="90" width="20" height="60"/>
        <text x="90" y="165">West</text>
        <text x="90" y="85">6</text>
    </g>
</svg>
```

The following stylesheet transforms the data into VRML:

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

<!-- generate text output as mime type model/vrml, using default charset -->
<xsl:output method="text" encoding="UTF-8" media-type="model/vrml"/>

        <xsl:template match="/">#VRML V2.0 utf8

# externproto definition of a single bar element
EXTERNPROTO bar [
  field SFInt32 x
  field SFInt32 y
  field SFInt32 z
  field SFString name
  ]
  "http://www.vrml.org/WorkingGroups/dbwork/barProto.wrl"

# inline containing the graph axes
Inline {
        url "http://www.vrml.org/WorkingGroups/dbwork/barAxes.wrl"
        }

                <xsl:for-each select="sales/division">
bar {
        x <xsl:value-of select="revenue"/>
        y <xsl:value-of select="growth"/>
        z <xsl:value-of select="bonus"/>
        name "<xsl:value-of select="@id"/>"
        }
```

```
                </xsl:for-each>

        </xsl:template>

</xsl:stylesheet>
```

The VRML output is:

```
#VRML V2.0 utf8

# externproto definition of a single bar element
EXTERNPROTO bar [
  field SFInt32 x
  field SFInt32 y
  field SFInt32 z
  field SFString name
  ]
  "http://www.vrml.org/WorkingGroups/dbwork/barProto.wrl"

# inline containing the graph axes
Inline {
        url "http://www.vrml.org/WorkingGroups/dbwork/barAxes.wrl"
        }


bar {
        x 10
        y 9
        z 7
        name "North"
        }

bar {
        x 4
        y 3
        z 4
        name "South"
        }

bar {
        x 6
        y -1.5
        z 2
        name "West"
        }
```

E Acknowledgements (Non-Normative)

The following have contributed to authoring this draft:

- Daniel Lipkin, Saba
- Jonathan Marsh, Microsoft
- Henry Thompson, University of Edinburgh
- Norman Walsh, Arbortext
- Steve Zilles, Adobe

This specification was developed and approved for publication by the W3C XSL Working Group (WG).

WG approval of this specification does not necessarily imply that all WG members voted for its approval. The current members of the XSL WG are:

Sharon Adler, IBM (Co-Chair); Anders Berglund, IBM; Perin Blanchard, Novell; Scott Boag, Lotus; Larry Cable, Sun; Jeff Caruso, Bitstream; James Clark; Peter Danielsen, Bell Labs; Don Day, IBM; Stephen Deach, Adobe; Dwayne Dicks, SoftQuad; Andrew Greene, Bitstream; Paul Grosso, Arbortext; Eduardo Gutentag, Sun; Juliane Harbarth, Software AG; Mickey Kimchi, Enigma; Chris Lilley, W3C; Chris Maden, Exemplary Technologies; Jonathan Marsh, Microsoft; Alex Milowski, Lexica; Steve Muench, Oracle; Scott Parnell, Xerox; Vincent Quint, W3C; Dan Rapp, Novell; Gregg Reynolds, Datalogics; Jonathan Robie, Software AG; Mark Scardina, Oracle; Henry Thompson, University of Edinburgh; Philip Wadler, Bell Labs; Norman Walsh, Arbortext; Sanjiva Weerawarana, IBM; Steve Zilles, Adobe (Co-Chair)

F Changes from Last Call Working Draft (Non-Normative)

The following are the changes since the previous public working draft:

- The XSLT namespace URI has been changed to `http://www.w3.org/1999/XSL/Transform`. The version number is now specified by `version` and `xsl:version` attributes.

- The mechanism for quoting namespace URIs has been replaced by a new namespace aliasing mechanism (see [**7.1.1 Literal Result Elements**]).

- The **system-property** no longer treats property names without a namespace URI specially.

- The `html` output method should not escape < characters occurring in attribute values.

- The `html` output method should not output a document type declaration if neither the `doctype-public` nor the `doctype-system` attributes are specified.

- It is an error for a top-level match pattern or a `use` expression on `xsl:key` to contain a variable reference.

- The expression in a `use` attribute on `xsl:key` is not restricted to return a node-set (see [**12.2 Keys**]).

- The `extension-element-available` and `extension-function-available` functions have been renamed to **element-available** and **function-available** and now work for XSLT defined elements and functions, as well as extension elements and functions. Similarly, `xsl:fallback` now works for XSLT defined elements, as well as extension elements. See [**15 Fallback**].

- The **document** function does not ignore its second argument, even if its first argument is of type node-set.

- The `xsl:locale` element has been renamed to `xsl:decimal-format`.

- The default value for the `NaN` attribute on `xsl:decimal-format` has been changed to the string

`NaN`.

- The default value for the `infinity` attribute on `xsl:decimal-format` has been changed to the string `Infinity`.
- With `data-type="number"`, `xsl:sort` ignores the `lang` attribute.
- Patterns may use the full `child::` and `attribute::` syntax in addition to the abbreviated syntax (see [**5.2 Patterns**]).
- The `xml-declaration` attribute on the `xsl:output` element has been renamed to `omit-xml-declaration` and its semantics inverted.
- The value of `other` for the `letter-value` attribute of `xsl:number` has been renamed to `traditional` (see [**7.7.1 Number to String Conversion Attributes**]).
- The `exclude-result-prefixes` attribute on the `xsl:stylesheet` element and the `xsl:exclude-result-prefixes` attribute on literal result elements can be used to avoid superfluous namespace declarations in the result tree (see [**7.1.1 Literal Result Elements**]).
- A `terminate` attribute has been added to `xsl:message` (see [**13 Messages**]).

G Features under Consideration for Future Versions of XSLT (Non-Normative)

The following features are under consideration for versions of XSLT after XSLT 1.0:

- a conditional expression;
- support for XML Schema datatypes and archetypes;
- support for something like style rules in the original XSL submission;
- an attribute to control the default namespace for names occurring in XSLT attributes;
- support for entity references;
- support for DTDs in the data model;
- support for notations in the data model;
- a way to get back from an element to the elements that reference it (e.g. by IDREF attributes);
- an easier way to get an ID or key in another document;
- support for regular expressions for matching against any or all of text nodes, attribute values, attribute names, element type names;

- case-insensitive comparisons;
- normalization of strings before comparison, for example for compatibility characters;
- a function `string resolve(node-set)` function that treats the value of the argument as a relative URI and turns it into an absolute URI using the base URI of the node;
- multiple result documents;
- defaulting the `select` attribute on `xsl:value-of` to the current node;
- an attribute on `xsl:attribute` to control how the attribute value is normalized;
- additional attributes on `xsl:sort` to provide further control over sorting, such as relative order of scripts;
- a way to put the text of a resource identified by a URI into the result tree;
- allow unions in steps (e.g. `foo/(bar|baz)`);
- allow for result tree fragments all operations that are allowed for node-sets;
- a way to group together consecutive nodes having duplicate subelements or attributes;
- features to make handling of the HTML `style` attribute more convenient.