XSL Transformations (XSLT) Version 2.0

W3C Recommendation 23 January 2007

This version:
http://www.w3.org/TR/2007/REC-xslt20-20070123/
Latest version:
http://www.w3.org/TR/xslt20/
Previous version:
http://www.w3.org/TR/2006/PR-xslt20-20061121/
Editor:
Michael Kay, Saxonica <http://www.saxonica.com/>

Please refer to the **errata** for this document, which may include some normative corrections.

See also **translations**.

Abstract

This specification defines the syntax and semantics of XSLT 2.0, a language for transforming XML documents into other XML documents.

XSLT 2.0 is a revised version of the XSLT 1.0 Recommendation [XSLT 1.0] published on 16 November 1999.

XSLT 2.0 is designed to be used in conjunction with XPath 2.0, which is defined in [XPath 2.0]. XSLT shares the same data model as XPath 2.0, which is defined in [Data Model], and it uses the library of functions and operators defined in [Functions and Operators].

XSLT 2.0 also includes optional facilities to serialize the results of a transformation, by means of an interface to the serialization component described in [XSLT and XQuery Serialization].

This document contains hyperlinks to specific sections or definitions within other documents in this family of specifications. These links are indicated visually by a superscript identifying the target specification: for example XP for XPath, DM for the XDM data model, FO for Functions and Operators.

Status of this Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at http://www.w3.org/TR/.

This Recommendation builds on the success of [XSLT 1.0], which was published on 16 November 1999. Many new features have been added to the language (see *J.2 New Functionality*) while retaining a high level of backwards compatibility (see *J.1 Incompatible Changes*). The changes have been designed to meet the requirements for XSLT 2.0 described in [XSLT 2.0 Requirements]. The way in which each requirement has been addressed is outlined in *I Checklist of Requirements*.

XSLT 2.0 depends on a number of other specifications that have progressed to Recommendation status at the same time: see [XPath 2.0], [Data Model], [Functions and Operators], and [XSLT and XQuery Serialization]. These subsidiary documents are also referenced in the specification of XQuery 1.0.

This document has been produced by the XSL Working Group, which is part of the XML Activity. The document has been reviewed by W3C Members and other interested parties, and is endorsed by the Director. It is a stable document and may be used as reference material or cited as a normative reference from another document. W3C's role in making the Recommendation is to draw attention to the specification and to promote its widespread deployment. This enhances the functionality and interoperability of the Web.

A small number of editorial corrections and clarifications have been made to the document since it was published as a Proposed Recommendation on 21 November 2006. These changes are listed at *J.2.4 Changes since Proposed Recommendation*.

Please record any comments about this document in W3C's public Bugzilla system (instructions can be found at http://www.w3.org/XML/2005/04/qt-bugzilla). If access to that system is not feasible, you may send your comments to the W3C XSLT/XPath/XQuery public comments mailing list, public-qt-comments@w3.org. It is helpful to include the string [XSLT] in the subject line of your comment, whether made in Bugzilla or in email. Each Bugzilla entry and email message should contain only one comment. Archives of the comments and responses are available at http://lists.w3.org/Archives/Public/public-qt-comments/.

General public discussion of XSLT takes place on the XSL-List forum.

This document was produced by a group operating under the 5 February 2004 W3C Patent Policy. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

Table of Contents

---

1 Introduction

1.1 What is XSLT?

This specification defines the syntax and semantics of the XSLT 2.0 language.

[DEFINITION: A transformation in the XSLT language is expressed in the form of a **stylesheet**, whose syntax is well-formed XML [XML 1.0] conforming to the Namespaces in XML Recommendation [Namespaces in XML 1.0].]

A stylesheet generally includes elements that are defined by XSLT as well as elements that are not defined by XSLT. XSLT-defined elements are distinguished by use of the namespace `http://www.w3.org/1999/XSL/Transform` (see *3.1 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.

The term stylesheet reflects the fact that one of the important roles of XSLT is to add styling information to an XML source document, by transforming it into a document consisting of XSL formatting objects (see [Extensible Stylesheet Language (XSL)]), or into another presentation-oriented format such as HTML, XHTML, or SVG. However, XSLT is used for a wide range of transformation tasks, not exclusively for formatting and presentation applications.

A transformation expressed in XSLT describes rules for transforming zero or more source trees into one or more result trees. The structure of these trees is described in [Data Model]. The transformation is achieved by a set of template rules. A template rule associates a pattern, which matches nodes in the source document, with a sequence constructor. In many cases, evaluating the sequence constructor will cause new nodes to be constructed, which can be used to produce part of a result tree. The structure of the result trees can be completely different from the structure of the source trees. In constructing a result tree, nodes from the source trees can be filtered and reordered, and arbitrary structure can be added. This mechanism allows a stylesheet to be applicable to a wide class of documents that have similar source tree structures.

[DEFINITION: A stylesheet may consist of several stylesheet modules, contained in different XML documents. For a given transformation, one of these functions as the **principal stylesheet module**. The complete stylesheet is assembled by finding the stylesheet modules referenced directly or indirectly from the principal stylesheet module using `xsl:include` and `xsl:import` elements: see *3.10.2 Stylesheet Inclusion* and *3.10.3 Stylesheet Import*.]

1.2 What's New in XSLT 2.0?

XSLT 1.0 was published in November 1999, and version 2.0 represents a significant increase in the capability of the language. A detailed list of changes is included in *J Changes from XSLT 1.0*. XSLT 2.0 has been developed in parallel with XPath 2.0 (see [XPath 2.0]), so the changes to XPath must be considered alongside the changes to XSLT.

2 Concepts

2.1 Terminology

For a full glossary of terms, see *C Glossary*.

[DEFINITION: The software responsible for transforming source trees into result trees using an XSLT stylesheet is referred to as the **processor**. This is sometimes expanded to *XSLT processor* to avoid any confusion with other processors, for example an XML processor.]

[DEFINITION: A specific product that performs the functions of an XSLT processor is referred to as an **implementation**].

[DEFINITION: The term **result tree** is used to refer to any tree constructed by instructions in the stylesheet. A result tree is either a final result

tree or a temporary tree.]

[DEFINITION: A **final result tree** is a result tree that forms part of the final output of a transformation. Once created, the contents of a final result tree are not accessible within the stylesheet itself.] The `xsl:result-document` instruction always creates a final result tree, and a final result tree may also be created implicitly by the initial template. The conditions under which this happens are described in *2.4 Executing a Transformation*. A final result tree MAY be serialized as described in *20 Serialization*.

[DEFINITION: The term **source tree** means any tree provided as input to the transformation. This includes the document containing the initial context node if any, documents containing nodes supplied as the values of stylesheet parameters, documents obtained from the results of functions such as `document`, `doc` [FO], and `collection` [FO], and documents returned by extension functions or extension instructions. In the context of a particular XSLT instruction, the term **source tree** means any tree provided as input to that instruction; this may be a source tree of the transformation as a whole, or it may be a temporary tree produced during the course of the transformation.]

[DEFINITION: The term **temporary tree** means any tree that is neither a source tree nor a final result tree.] Temporary trees are used to hold intermediate results during the execution of the transformation.

In this specification the phrases MUST, MUST NOT, SHOULD, SHOULD NOT, MAY, REQUIRED, and RECOMMENDED are to be interpreted as described in [RFC2119].

Where the phrase MUST, MUST NOT, or REQUIRED relates to the behavior of the XSLT processor, then an implementation is not conformant unless it behaves as specified, subject to the more detailed rules in *21 Conformance*.

Where the phrase MUST, MUST NOT, or REQUIRED relates to a stylesheet, then the processor MUST enforce this constraint on stylesheets by reporting an error if the constraint is not satisfied.

Where the phrase SHOULD, SHOULD NOT, or RECOMMENDED relates to a stylesheet, then a processor MAY produce warning messages if the constraint is not satisfied, but MUST NOT treat this as an error.

[DEFINITION: In this specification, the term **implementation-defined** refers to a feature where the implementation is allowed some flexibility, and where the choices made by the implementation MUST be described in documentation that accompanies any conformance claim.]

[DEFINITION: The term **implementation-dependent** refers to a feature where the behavior MAY vary from one implementation to another, and where the vendor is not expected to provide a full specification of the behavior.] (This might apply, for example, to limits on the size of source documents that can be transformed.)

In all cases where this specification leaves the behavior implementation-defined or implementation-dependent, the implementation has the option of providing mechanisms that allow the user to influence the behavior.

A paragraph labeled as a **Note** or described as an **example** is non-normative.

Many terms used in this document are defined in the XPath specification [XPath 2.0] or the XDM specification [Data Model]. Particular attention is drawn to the following:

- [DEFINITION: The term **atomization** is defined in Section 2.4.2 Atomization [XP]. It is a process that takes as input a sequence of nodes and atomic values, and returns a sequence of atomic values, in which the nodes are replaced by their typed values as defined in [Data Model].] For some nodes (for example, elements with element-only content), atomization generates a dynamic error.
- [DEFINITION: The term **typed value** is defined in Section 5.15 typed-value Accessor [DM]. Every node except an element defined in the schema with element-only content has a typed value. For example, the typed value of an attribute of type `xs:IDREFS` is a sequence of zero or more `xs:IDREF` values.]
- [DEFINITION: The term **string value** is defined in Section 5.13 string-value Accessor [DM]. Every node has a string value. For example, the string value of an element is the concatenation of the string values of all its descendant text nodes.]
- [DEFINITION: The term **XPath 1.0 compatibility mode** is defined in Section 2.1.1 Static Context [XP]. This is a setting in the static context of an XPath expression; it has two values, `true` and `false`. When the value is set to true, the semantics of function calls and certain other operations are adjusted to give a greater degree of backwards compatibility between XPath 2.0 and XPath 1.0.]

[DEFINITION: The term **core function** means a function that is specified in [Functions and Operators] and that is in the standard function namespace.]

2.2 Notation

[DEFINITION: An **XSLT element** is an element in the XSLT namespace whose syntax and semantics are defined in this specification.] For a non-normative list of XSLT elements, see *D Element Syntax Summary*.

In this document the specification of each XSLT element is preceded by a summary of its syntax in the form of a model for elements of that element type. A full list of all these specifications can be found in *D Element Syntax Summary*. The meaning of syntax summary notation is as follows:

- An attribute that is REQUIRED is shown with its name in bold. An attribute that may be omitted is shown with a question mark following its name.
- An attribute that is deprecated is shown in a grayed font within square brackets.
- The string that occurs in the place of an attribute value specifies the allowed values of the attribute. If this is surrounded by curly brackets (`{...}`), then the attribute value is treated as an attribute value template, and the string occurring within curly brackets specifies the allowed values of the result of evaluating 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.

  In all cases where this specification states that the value of an attribute MUST be one of a limited set of values, leading and trailing whitespace in the attribute value is ignored. In the case of an attribute value template, this applies to the effective value obtained when the attribute value template is expanded.
- Unless the element is REQUIRED to be empty, the model element contains a comment specifying the allowed content. The allowed content is specified in a similar way to an element type declaration in XML; *sequence constructor* means that any mixture of text nodes, literal result elements, extension instructions, and XSLT elements from the instruction category is allowed; *other-declarations* means that any mixture of XSLT elements from the declaration category, other than `xsl:import`, is allowed, together with user-defined data elements.

- The element is prefaced by comments indicating if it belongs to the `instruction` category or `declaration` category or both. The category of an element only affects whether it is allowed in the content of elements that allow a sequence constructor or *other-declarations*.

Example: Syntax Notation

This example illustrates the notation used to describe XSLT elements.

```
<!-- Category: instruction -->
<xsl:example-element
  select = expression
  debug? = { "yes" | "no" }>
  <!-- Content: ((xsl:variable | xsl:param)*, xsl:sequence) -->
</xsl:example-element>
```

This example defines a (non-existent) element `xsl:example-element`. The element is classified as an instruction. It takes a mandatory `select` attribute, whose value is an XPath expression, and an optional `debug` attribute, whose value MUST be either `yes` or `no`; the curly brackets indicate that the value can be defined as an attribute value template, allowing a value such as `debug="{$debug}"`, where the variable `debug` is evaluated to yield `"yes"` or `"no"` at run-time.

The content of an `xsl:example-element` instruction is defined to be a sequence of zero or more `xsl:variable` and `xsl:param` elements, followed by an `xsl:sequence` element.

[ERR XTSE0010] A static error is signaled if an XSLT-defined element is used in a context where it is not permitted, if a REQUIRED attribute is omitted, or if the content of the element does not correspond to the content that is allowed for the element.

Attributes are validated as follows. These rules apply to the value of the attribute after removing leading and trailing whitespace.

- [ERR XTSE0020] It is a static error if an attribute (other than an attribute written using curly brackets in a position where an attribute value template is permitted) contains a value that is not one of the permitted values for that attribute.
- [ERR XTDE0030] It is a non-recoverable dynamic error if the effective value of an attribute written using curly brackets, in a position where an attribute value template is permitted, is a value that is not one of the permitted values for that attribute. If the processor is able to detect the error statically (for example, when any XPath expressions within the curly brackets can be evaluated statically), then the processor may optionally signal this as a static error.

Special rules apply if the construct appears in part of the stylesheet that is processed with forwards-compatible behavior: see *3.9 Forwards-Compatible Processing*.

[DEFINITION: Some constructs defined in this specification are described as being **deprecated**. The use of this term implies that stylesheet authors SHOULD NOT use the construct, and that the construct may be removed in a later version of this specification.] All constructs that are deprecated in this specification are also (as it happens) optional features that implementations are NOT REQUIRED to provide.

Note:

This working draft includes a non-normative XML Schema for XSLT stylesheet modules (see *G Schema for XSLT Stylesheets*). The syntax summaries described in this section are normative.

XSLT defines a set of standard functions which are additional to those defined in [Functions and Operators]. The signatures of these functions are described using the same notation as used in [Functions and Operators]. The names of these functions are all in the standard function namespace.

2.3 Initiating a Transformation

This document does not specify any application programming interfaces or other interfaces for initiating a transformation. This section, however, describes the information that is supplied when a transformation is initiated. Except where otherwise indicated, the information is REQUIRED.

Implementations MAY allow a transformation to run as two or more phases, for example parsing, compilation and execution. Such a distinction is outside the scope of this specification, which treats transformation as a single process controlled using a set of stylesheet modules, supplied in the form of XML documents.

The following information is supplied to execute a transformation:

- The stylesheet module that is to act as the principal stylesheet module for the transformation. The complete stylesheet is assembled by recursively expanding the `xsl:import` and `xsl:include` declarations in the principal stylesheet module, as described in *3.10.2 Stylesheet Inclusion* and *3.10.3 Stylesheet Import*.
- A set (possibly empty) of values for stylesheet parameters (see *9.5 Global Variables and Parameters*). These values are available for use within expressions in the stylesheet.
- [DEFINITION: A node that acts as the **initial context node** for the transformation. This node is accessible within the stylesheet as the initial value of the XPath expressions . (dot) and `self::node()`, as described in *5.4.3.1 Maintaining Position: the Focus*].

  If no initial context node is supplied, then the context item, context position, and context size will initially be undefined, and the evaluation of any expression that references these values will result in a dynamic error. (Note that the initial context size and context position will always be 1 (one) when an initial context node is supplied, and will be undefined if no initial context node is supplied).
- Optionally, the name of a named template which is to be executed as the entry point to the transformation. This template MUST exist within the stylesheet. If no named template is supplied, then the transformation starts with the template rule that best matches the initial context node, according to the rules defined in *6.4 Conflict Resolution for Template Rules*. Either a named template, or an initial context node, or both, MUST be supplied.
- Optionally, an initial mode. This MUST either be the default mode, or a mode that is explicitly named in the `mode` attribute of an `xsl:template` declaration within the stylesheet. If an initial mode is supplied, then in searching for the template rule that best matches the initial context node, the processor considers only those rules that apply to the initial mode. If no initial mode is supplied, the default mode is used.

- A base output URI. [DEFINITION: The **base output URI** is a URI to be used as the base URI when resolving a relative URI allocated to a final result tree. If the transformation generates more than one final result tree, then typically each one will be allocated a URI relative to this base URI.] The way in which a base output URI is established is implementation-defined.
- A mechanism for obtaining a document node and a media type, given an absolute URI. The total set of available documents (modeled as a mapping from URIs to document nodes) forms part of the context for evaluating XPath expressions, specifically the `doc` [FO] function. The XSLT `document` function additionally requires the media type of the resource representation, for use in interpreting any fragment identifier present within a URI Reference.

  Note:

  The set of documents that are available to the stylesheet is implementation-dependent, as is the processing that is carried out to construct a tree representing the resource retrieved using a given URI. Some possible ways of constructing a document (specifically, rules for constructing a document from an Infoset or from a PSVI) are described in [Data Model].

[ERR XTDE0040] It is a non-recoverable dynamic error if the invocation of the stylesheet specifies a template name that does not match the expanded-QName of a named template defined in the stylesheet.

[ERR XTDE0045] It is a non-recoverable dynamic error if the invocation of the stylesheet specifies an initial mode (other than the default mode) that does not match the expanded-QName in the `mode` attribute of any template defined in the stylesheet.

[ERR XTDE0047] It is a non-recoverable dynamic error if the invocation of the stylesheet specifies both an initial mode and an initial template.

[ERR XTDE0050] It is a non-recoverable dynamic error if the stylesheet that is invoked declares a visible stylesheet parameter with `required="yes"` and no value for this parameter is supplied during the invocation of the stylesheet. A stylesheet parameter is visible if it is not masked by another global variable or parameter with the same name and higher import precedence.

[DEFINITION: The transformation is performed by evaluating an **initial template**. If a named template is supplied when the transformation is initiated, then this is the initial template; otherwise, the initial template is the template rule selected according to the rules of the `xsl:apply-templates` instruction for processing the initial context node in the initial mode.]

Parameters passed to the transformation by the client application are matched against stylesheet parameters (see *9.5 Global Variables and Parameters*), not against the template parameters declared within the initial template. All template parameters within the initial template to be executed will take their default values.

[ERR XTDE0060] It is a non-recoverable dynamic error if the initial template defines a template parameter that specifies `required="yes"`.

A stylesheet can process further source documents in addition to those supplied when the transformation is invoked. These additional documents can be loaded using the functions `document` (see *16.1 Multiple Source Documents*) or `doc` [FO] or `collection` [FO] (see [Functions and Operators]), or they can be supplied as stylesheet parameters (see *9.5 Global Variables and Parameters*), or as the result of an extension function (see *18.1 Extension Functions*).

2.4 Executing a Transformation

[DEFINITION: A stylesheet contains a set of **template rules** (see *6 Template Rules*). A template rule has three parts: a pattern that is matched against nodes, a (possibly empty) set of template parameters, and a sequence constructor that is evaluated to produce a sequence of items.] In many cases these items are newly constructed nodes, which are then written to a result tree.

A transformation as a whole is executed by evaluating the sequence constructor of the initial template as described in *5.7 Sequence Constructors*.

If the initial template has an `as` attribute, then the result sequence of the initial template is checked against the required type in the same way as for any other template. If this result sequence is non-empty, then it is used to construct an implicit final result tree, following the rules described in *5.7.1 Constructing Complex Content*: the effect is as if the initial template *T* were called by an implicit template of the form:

```
<xsl:template name="IMPLICIT">
  <xsl:result-document href="">
    <xsl:call-template name="T"/>
  </xsl:result-document>
</xsl:template>
```

An implicit result tree is also created when the result sequence is empty, provided that no `xsl:result-document` instruction has been evaluated during the course of the transformation. In this situation the implicit result tree will consist of a document node with no children.

Note:

This means that there is always at least one result tree. It also means that if the content of the initial template is a single `xsl:result-document` instruction, as in the example above, then only one result tree is produced, not two. It is useful to make the result document explicit as this is the only way of invoking document-level validation.

If the result of the initial template is non-empty, and an explicit `xsl:result-document` instruction has been evaluated with the empty attribute `href=""`, then an error will occur [see ERR XTDE1490], since it is not possible to create two final result trees with the same URI.

A sequence constructor is a sequence of sibling nodes in the stylesheet, each of which is either an XSLT instruction, a literal result element, a text node, or an extension instruction.

[DEFINITION: An **instruction** is either an XSLT instruction or an extension instruction.]

[DEFINITION: An **XSLT instruction** is an XSLT element whose syntax summary in this specification contains the annotation `<!-- category: instruction -->`.]

Extension instructions are described in *18.2 Extension Instructions*.

The main categories of XSLT instruction are as follows:

- instructions that create new nodes: `xsl:document`, `xsl:element`, `xsl:attribute`, `xsl:processing-instruction`, `xsl:comment`,

xsl:value-of, xsl:text, xsl:namespace;

- an instruction that returns an arbitrary sequence by evaluating an XPath expression: xsl:sequence;
- instructions that cause conditional or repeated evaluation of nested instructions: xsl:if, xsl:choose, xsl:for-each, xsl:for-each-group;
- instructions that invoke templates: xsl:apply-templates, xsl:apply-imports, xsl:call-template, xsl:next-match;
- Instructions that declare variables: xsl:variable, xsl:param;
- other specialized instructions: xsl:number, xsl:analyze-string, xsl:message, xsl:result-document.

Often, a sequence constructor will include an xsl:apply-templates instruction, which selects a sequence of nodes to be processed. Each of the selected nodes is processed by searching the stylesheet for a matching template rule and evaluating the sequence constructor of that template rule. The resulting sequences of items are concatenated, in order, to give the result of the xsl:apply-templates instruction, as described in *6.3 Applying Template Rules*; this sequence is often added to a result tree. Since the sequence constructors of the selected template rules may themselves contain xsl:apply-templates instructions, this results in a cycle of selecting nodes, identifying template rules, constructing sequences, and constructing result trees, that recurses through a source tree.

2.5 The Evaluation Context

The results of some expressions and instructions in a stylesheet may depend on information provided contextually. This context information is divided into two categories: the static context, which is known during static analysis of the stylesheet, and the dynamic context, which is not known until the stylesheet is evaluated. Although information in the static context is known at analysis time, it is sometimes used during stylesheet evaluation.

Some context information can be set by means of declarations within the stylesheet itself. For example, the namespace bindings used for any XPath expression are determined by the namespace declarations present in containing elements in the stylesheet. Other information may be supplied externally or implicitly: an example is the current date and time.

The context information used in processing an XSLT stylesheet includes as a subset all the context information required when evaluating XPath expressions. The XPath 2.0 specification defines a static and dynamic context that the host language (in this case, XSLT) may initialize, which affects the results of XPath expressions used in that context. XSLT augments the context with additional information: this additional information is used firstly by XSLT constructs outside the scope of XPath (for example, the xsl:sort element), and secondly, by functions that are defined in the XSLT specification (such as key and format-number) that are available for use in XPath expressions appearing within a stylesheet.

The static context for an expression or other construct in a stylesheet is determined by the place in which it appears lexically. The details vary for different components of the static context, but in general, elements within a stylesheet module affect the static context for their descendant elements within the same stylesheet module.

The dynamic context is maintained as a stack. When an instruction or expression is evaluated, it may add dynamic context information to the stack; when evaluation is complete, the dynamic context reverts to its previous state. An expression that accesses information from the dynamic context always uses the value at the top of the stack.

The most commonly used component of the dynamic context is the context item. This is an implicit variable whose value is the item (it may be a node or an atomic value) currently being processed. The value of the context item can be referenced within an XPath expression using the expression . (dot).

Full details of the static and dynamic context are provided in *5.4 The Static and Dynamic Context*.

2.6 Parsing and Serialization

An XSLT stylesheet describes a process that constructs a set of final result trees from a set of source trees.

The stylesheet does not describe how a source tree is constructed. Some possible ways of constructing source trees are described in [Data Model]. Frequently an implementation will operate in conjunction with an XML parser (or more strictly, in the terminology of [XML 1.0], an *XML processor*), to build a source tree from an input XML document. An implementation MAY also provide an application programming interface allowing the tree to be constructed directly, or allowing it to be supplied in the form of a DOM Document object (see [DOM Level 2]). This is outside the scope of this specification. Users should be aware, however, that since the input to the transformation is a tree conforming to the XDM data model as described in [Data Model], constructs that might exist in the original XML document, or in the DOM, but which are not within the scope of the data model, cannot be processed by the stylesheet and cannot be guaranteed to remain unchanged in the transformation output. Such constructs include CDATA section boundaries, the use of entity references, and the DOCTYPE declaration and internal DTD subset.

[DEFINITION: A frequent requirement is to output a final result tree as an XML document (or in other formats such as HTML). This process is referred to as **serialization**.]

Like parsing, serialization is not part of the transformation process, and it is not REQUIRED that an XSLT processor MUST be able to perform serialization. However, for pragmatic reasons, this specification describes declarations (the xsl:output element and the xsl:character-map declarations, see *20 Serialization*), and attributes on the xsl:result-document instruction, that allow a stylesheet to specify the desired properties of a serialized output file. When serialization is not being performed, either because the implementation does not support the serialization option, or because the user is executing the transformation in a way that does not invoke serialization, then the content of the xsl:output and xsl:character-map declarations has no effect. Under these circumstances the processor MAY report any errors in an xsl:output or xsl:character-map declaration, or in the serialization attributes of xsl:result-document, but is not REQUIRED to do so.

2.7 Extensibility

XSLT defines a number of features that allow the language to be extended by implementers, or, if implementers choose to provide the capability, by users. These features have been designed, so far as possible, so that they can be used without sacrificing interoperability. Extensions other than those explicitly defined in this specification are not permitted.

These features are all based on XML namespaces; namespaces are used to ensure that the extensions provided by one implementer do not clash with those of a different implementer.

The most common way of extending the language is by providing additional functions, which can be invoked from XPath expressions. These are known as extension functions, and are described in *18.1 Extension Functions*.

It is also permissible to extend the language by providing new instructions. These are referred to as extension instructions, and are described in *18.2 Extension Instructions*. A stylesheet that uses extension instructions must declare that it is doing so by using the `[xsl:]extension-element-prefixes` attribute.

Extension instructions and extension functions defined according to these rules MAY be provided by the implementer of the XSLT processor, and the implementer MAY also provide facilities to allow users to create further extension instructions and extension functions.

This specification defines how extension instructions and extension functions are invoked, but the facilities for creating new extension instructions and extension functions are implementation-defined. For further details, see *18 Extensibility and Fallback*.

The XSLT language can also be extended by the use of extension attributes (see *3.3 Extension Attributes*), and by means of user-defined data elements (see *3.6.2 User-defined Data Elements*).

2.8 Stylesheets and XML Schemas

An XSLT stylesheet can make use of information from a schema. An XSLT transformation can take place in the absence of a schema (and, indeed, in the absence of a DTD), but where the source document has undergone schema validity assessment, the XSLT processor has access to the type information associated with individual nodes, not merely to the untyped text.

Information from a schema can be used both statically (when the stylesheet is compiled), and dynamically (during evaluation of the stylesheet to transform a source document).

There are places within a stylesheet, and within XPath expressions and patterns in a stylesheet, where it is possible to refer to named type definitions in a schema, or to element and attribute declarations. For example, it is possible to declare the types expected for the parameters of a function. This is done using the SequenceType [XP] syntax defined in [XPath 2.0].

[DEFINITION: Type definitions and element and attribute declarations are referred to collectively as **schema components**.]

[DEFINITION: The schema components that may be referenced by name in a stylesheet are referred to as the **in-scope schema components**. This set is the same throughout all the modules of a stylesheet.]

The conformance rules for XSLT 2.0, defined in *21 Conformance*, distinguish between a basic XSLT processor and a schema-aware XSLT processor. As the names suggest, a basic XSLT processor does not support the features of XSLT that require access to schema information, either statically or dynamically. A stylesheet that works with a basic XSLT processor will produce the same results with a schema-aware XSLT processor provided that the source documents are untyped (that is, they are not validated against a schema). However, if source documents are validated against a schema then the results may be different from the case where they are not validated. Some constructs that work on untyped data may fail with typed data (for example, an attribute of type `xs:date` cannot be used as an argument of the `substring` [FO] function) and other constructs may produce different results depending on the data type (for example, given the element `<product price="10.00" discount="2.00"/>`, the expression `@price gt @discount` will return true if the attributes have type `xs:decimal`, but will return false if they are untyped).

There is a standard set of type definitions that are always available as in-scope schema components in every stylesheet. These are defined in *3.13 Built-in Types*. The set of built-in types varies between a basic XSLT processor and a schema-aware XSLT processor.

The remainder of this section describes facilities that are available only with a schema-aware XSLT processor.

Additional schema components (type definitions, element declarations, and attribute declarations) may be added to the in-scope schema components by means of the `xsl:import-schema` declaration in a stylesheet.

The `xsl:import-schema` declaration may reference an external schema document by means of a URI, or it may contain an inline `xs:schema` element.

It is only necessary to import a schema explicitly if one or more of its schema components are referenced explicitly by name in the stylesheet; it is not necessary to import a schema merely because the stylesheet is used to process a source document that has been assessed against that schema. It is possible to make use of the information resulting from schema assessment (for example, the fact that a particular attribute holds a date) even if no schema has been imported by the stylesheet.

Further, importing a schema does not of itself say anything about the type of the source document that the stylesheet is expected to process. The imported type definitions can be used for temporary nodes or for nodes on a result tree just as much as for nodes in source documents. It is possible to make assertions about the type of an input document by means of tests within the stylesheet. For example:

Example: Asserting the Required Type of the Source Document

```
<xsl:template match="document-node(schema-element(my:invoice))" priority="2">
. . .
</xsl:template>

<xsl:template match="document-node()" priority="1">
  <xsl:message terminate="yes">Source document is not an invoice</xsl:message>
</xsl:template>
```

This example will cause the transformation to fail with an error message unless the document element of the source document is valid against the top-level element declaration `my:invoice`, and has been annotated as such.

It is possible that a source document may contain nodes whose type annotation is not one of the types imported by the stylesheet. This creates a potential problem because in the case of an expression such as `data(.) instance of xs:integer` the system needs to know whether the type named in the type annotation of the context node is derived by restriction from the type `xs:integer`. This information is not explicitly available in an XDM tree, as defined in [Data Model]. The implementation may choose one of several strategies for dealing with this situation:

1. The processor may signal a non-recoverable dynamic error if a source document is found to contain a type annotation that is not known to the processor.

2. The processor may maintain additional metadata, beyond that described in [Data Model], that allows the source document to be processed as if all the necessary schema information had been imported using `xsl:import-schema`. Such metadata might be held in the data structure representing the source document itself, or it might be held in a system catalog or repository.
3. The processor may be configured to use a fixed set of schemas, which are automatically used to validate all source documents before they can be supplied as input to a transformation. In this case it is impossible for a source document to have a type annotation that the processor is not aware of.
4. The processor may be configured to treat the source document as if no schema processing had been performed, that is, effectively to strip all type annotations from elements and attributes on input, marking them instead as having type `xs:untyped` and `xs:untypedAtomic` respectively.

Where a stylesheet author chooses to make assertions about the types of nodes or of variables and parameters, it is possible for an XSLT processor to perform static analysis of the stylesheet (that is, analysis in the absence of any source document). Such analysis MAY reveal errors that would otherwise not be discovered until the transformation is actually executed. An XSLT processor is not REQUIRED to perform such static type-checking. Under some circumstances (see *2.9 Error Handling*) type errors that are detected early MAY be reported as static errors. In addition an implementation MAY report any condition found during static analysis as a warning, provided that this does not prevent the stylesheet being evaluated as described by this specification.

A stylesheet can also control the type annotations of nodes that it constructs in a final result tree, or in temporary trees. This can be done in a number of ways.

- It is possible to request explicit validation of a complete document, that is, a tree rooted at a document node. This applies both to temporary trees constructed using the `xsl:document` (or `xsl:copy`) instruction and also to final result trees constructed using `xsl:result-document`. Validation is either strict or lax, as described in [XML Schema Part 1]. If validation of a result tree fails (strictly speaking, if the outcome of the validity assessment is `invalid`), then the transformation fails, but in all other cases, the element and attribute nodes of the tree will be annotated with the names of the types to which these nodes conform. These type annotations will be discarded if the result tree is serialized as an XML document, but they remain available when the result tree is passed to an application (perhaps another stylesheet) for further processing.
- It is also possible to validate individual element and attribute nodes as they are constructed. This is done using the `type` and `validation` attributes of the `xsl:element`, `xsl:attribute`, `xsl:copy`, and `xsl:copy-of` instructions, or the `xsl:type` and `xsl:validation` attributes of a literal result element.
- When elements, attributes, or document nodes are copied, either explicitly using the `xsl:copy` or `xsl:copy-of` instructions, or implicitly when nodes in a sequence are attached to a new parent node, the options `validation="strip"` and `validation="preserve"` are available, to control whether existing type annotations are to be retained or not.

When nodes in a temporary tree are validated, type information is available for use by operations carried out on the temporary tree, in the same way as for a source document that has undergone schema assessment.

For details of how validation of element and attribute nodes works, see *19.2 Validation*.

2.9 Error Handling

[DEFINITION: An error that is detected by examining a stylesheet before execution starts (that is, before the source document and values of stylesheet parameters are available) is referred to as a **static error**.]

Errors classified in this specification as static errors MUST be signaled by all implementations: that is, the processor MUST indicate that the error is present. A static error MUST be signaled even if it occurs in a part of the stylesheet that is never evaluated. Static errors are never recoverable. After signaling a static error, a processor MAY continue for the purpose of signaling additional errors, but it MUST eventually terminate abnormally without producing any final result tree.

There is an exception to this rule when the stylesheet specifies forwards-compatible behavior (see *3.9 Forwards-Compatible Processing*).

Generally, errors in the structure of the stylesheet, or in the syntax of XPath expressions contained in the stylesheet, are classified as static errors. Where this specification states that an element in the stylesheet MUST or MUST NOT appear in a certain position, or that it MUST or MUST NOT have a particular attribute, or that an attribute MUST or MUST NOT have a value satisfying specified conditions, then any contravention of this rule is a static error unless otherwise specified.

[DEFINITION: An error that is not detected until a source document is being transformed is referred to as a **dynamic error**.]

[DEFINITION: Some dynamic errors are classed as **recoverable errors**. When a recoverable error occurs, this specification allows the processor either to signal the error (by reporting the error condition and terminating execution) or to take a defined recovery action and continue processing.] It is implementation-defined whether the error is signaled or the recovery action is taken.

[DEFINITION: If an implementation chooses to recover from a recoverable dynamic error, it MUST take the **optional recovery action** defined for that error condition in this specification.]

When the implementation makes the choice between signaling a dynamic error or recovering, it is not restricted in how it makes the choice; for example, it MAY provide options that can be set by the user. When an implementation chooses to recover from a dynamic error, it MAY also take other action, such as logging a warning message.

[DEFINITION: A dynamic error that is not recoverable is referred to as a **non-recoverable dynamic error**. When a non-recoverable dynamic error occurs, the processor MUST signal the error, and the transformation fails.]

Because different implementations may optimize execution of the stylesheet in different ways, the detection of dynamic errors is to some degree implementation-dependent. In cases where an implementation is able to produce the final result trees without evaluating a particular construct, the implementation is never REQUIRED to evaluate that construct solely in order to determine whether doing so causes a dynamic error. For example, if a variable is declared but never referenced, an implementation MAY choose whether or not to evaluate the variable declaration, which means that if evaluating the variable declaration causes a dynamic error, some implementations will signal this error and others will not.

There are some cases where this specification requires that a construct MUST NOT be evaluated: for example, the content of an `xsl:if` instruction MUST NOT be evaluated if the test condition is false. This means that an implementation MUST NOT signal any dynamic errors that would arise if the construct were evaluated.

An implementation MAY signal a dynamic error before any source document is available, but only if it can determine that the error would be

signaled for every possible source document and every possible set of parameter values. For example, some circularity errors fall into this category: see *9.8 Circular Definitions*.

The XPath specification states (see Section 2.3.1 Kinds of Errors[XP]) that if any expression (at any level) can be evaluated during the analysis phase (because all its explicit operands are known and it has no dependencies on the dynamic context), then any error in performing this evaluation MAY be reported as a static error. For XPath expressions used in an XSLT stylesheet, however, any such errors MUST NOT be reported as static errors in the stylesheet unless they would occur in every possible evaluation of that stylesheet; instead, they must be signaled as dynamic errors, and signaled only if the XPath expression is actually evaluated.

Example: Errors in Constant Subexpressions

An XPath processor may report statically that the expression `1 div 0` fails with a "divide by zero" error. But suppose this XPath expression occurs in an XSLT construct such as:

```
<xsl:choose>
  <xsl:when test="system-property('xsl:version') = '1.0'">
    <xsl:value-of select="1 div 0"/>
  </xsl:when>
  <xsl:otherwise>
    <xsl:value-of select="xs:double('INF')"/>
  </xsl:otherwise>
</xsl:choose>
```

Then the XSLT processor must not report an error, because the relevant XPath construct appears in a context where it will never be executed by an XSLT 2.0 processor. (An XSLT 1.0 processor will execute this code successfully, returning positive infinity, because it uses double arithmetic rather than decimal arithmetic.)

[DEFINITION: Certain errors are classified as **type errors**. A type error occurs when the value supplied as input to an operation is of the wrong type for that operation, for example when an integer is supplied to an operation that expects a node.] If a type error occurs in an instruction that is actually evaluated, then it MUST be signaled in the same way as a non-recoverable dynamic error. Alternatively, an implementation MAY signal a type error during the analysis phase in the same way as a static error, even if it occurs in part of the stylesheet that is never evaluated, provided it can establish that execution of a particular construct would never succeed.

It is implementation-defined whether type errors are signaled statically.

Example: A Type Error

The following construct contains a type error, because `42` is not allowed as an operand of the `xsl:apply-templates` instruction. An implementation MAY optionally signal this as a static error, even though the offending instruction will never be evaluated, and the type error would therefore never be signaled as a dynamic error.

```
<xsl:if test="false()">
  <xsl:apply-templates select="42"/>
</xsl:if>
```

On the other hand, in the following example it is not possible to determine statically whether the operand of `xsl:apply-templates` will have a suitable dynamic type. An implementation MAY produce a warning in such cases, but it MUST NOT treat it as an error.

```
<xsl:template match="para">
  <xsl:param name="p" as="item()"/>
  <xsl:apply-templates select="$p"/>
</xsl:template>
```

If more than one error arises, an implementation is not REQUIRED to signal any errors other than the first one that it detects. It is implementation-dependent which of the several errors is signaled. This applies both to static errors and to dynamic errors. An implementation is allowed to signal more than one error, but if any errors have been signaled, it MUST NOT finish as if the transformation were successful.

When a transformation signals one or more dynamic errors, the final state of any persistent resources updated by the transformation is implementation-dependent. Implementations are not REQUIRED to restore such resources to their initial state. In particular, where a transformation produces multiple result documents, it is possible that one or more serialized result documents MAY be written successfully before the transformation terminates, but the application cannot rely on this behavior.

Everything said above about error handling applies equally to errors in evaluating XSLT instructions, and errors in evaluating XPath expressions. Static errors and dynamic errors may occur in both cases.

[DEFINITION: If a transformation has successfully produced a final result tree, it is still possible that errors may occur in serializing the result tree. For example, it may be impossible to serialize the result tree using the encoding selected by the user. Such an error is referred to as a **serialization error**.] If the processor performs serialization, then it MUST do so as specified in *20 Serialization*, and in particular it MUST signal any serialization errors that occur.

Errors are identified by a QName. For errors defined in this specification, the namespace of the QName is always `http://www.w3.org/2005/xqt-errors` (and is therefore not given explicitly), while the local part is an 8-character code in the form *PPSSNNNN*. Here *PP* is always `XT` (meaning XSLT), and *SS* is one of `SE` (static error), `DE` (dynamic error), `RE` (recoverable dynamic error), or `TE` (type error). Note that the allocation of an error to one of these categories is purely for convenience and carries no normative implications about the way the error is handled. Many errors, for example, can be reported either dynamically or statically.

These error codes are used to label error conditions in this specification, and are summarized in *E Summary of Error Conditions*). They are provided primarily for ease of reference. Implementations MAY use these codes when signaling errors, but they are not REQUIRED to do so. An API specification, however, MAY require the use of error codes based on these QNames. Additional errors defined by an implementation

(or by an application) MAY use QNames in an implementation-defined (or user-defined) namespace without risk of collision.

Errors defined in the [XPath 2.0] and [Functions and Operators] specifications use QNames with a similar structure, in the same namespace. When errors occur in processing XPath expressions, an XSLT processor SHOULD use the original error code reported by the XPath processor, unless a more specific XSLT error code is available.

3 Stylesheet Structure

[DEFINITION: A stylesheet consists of one or more **stylesheet modules**, each one forming all or part of an XML document.]

Note:

A stylesheet module is represented by an XDM element node (see [Data Model]). In the case of a standard stylesheet module, this will be an `xsl:stylesheet` or `xsl:transform` element. In the case of a simplified stylesheet module, it can be any element (not in the XSLT namespace) that has an `xsl:version` attribute.

Although stylesheet modules will commonly be maintained in the form of documents conforming to XML 1.0 or XML 1.1, this specification does not mandate such a representation. As with source trees, the way in which stylesheet modules are constructed, from textual XML or otherwise, is outside the scope of this specification.

A stylesheet module is either a standard stylesheet module or a simplified stylesheet module:

- [DEFINITION: A **standard stylesheet module** is a tree, or part of a tree, consisting of an `xsl:stylesheet` or `xsl:transform` element (see *3.6 Stylesheet Element*) together with its descendant nodes and associated attributes and namespaces.]
- [DEFINITION: A **simplified stylesheet module** is a tree, or part of a tree, consisting of a literal result element together with its descendant nodes and associated attributes and namespaces. This element is not itself in the XSLT namespace, but it MUST have an `xsl:version` attribute, which implies that it MUST have a namespace node that declares a binding for the XSLT namespace. For further details see *3.7 Simplified Stylesheet Modules*.]

Both forms of stylesheet module (standard and simplified) can exist either as an entire XML document, or embedded as part of another XML document, typically but not necessarily a source document that is to be processed using the stylesheet.

[DEFINITION: A **standalone stylesheet module** is a stylesheet module that comprises the whole of an XML document.]

[DEFINITION: An **embedded stylesheet module** is a stylesheet module that is embedded within another XML document, typically the source document that is being transformed.] (see *3.11 Embedded Stylesheet Modules*).

There are thus four kinds of stylesheet module:

standalone standard stylesheet modules
standalone simplified stylesheet modules
embedded standard stylesheet modules
embedded simplified stylesheet modules

3.1 XSLT Namespace

[DEFINITION: The **XSLT namespace** has the URI `http://www.w3.org/1999/XSL/Transform`. It is used to identify elements, attributes, and other names that have a special meaning defined in this specification.]

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 *3.6 Stylesheet Element* and *3.7 Simplified Stylesheet Modules*).

XSLT processors MUST use the XML namespaces mechanism [Namespaces in XML 1.0] to recognize elements and attributes from this namespace. Elements from the XSLT namespace are recognized only in the stylesheet and not in the source document. The complete list of XSLT-defined elements is specified in *D Element Syntax Summary*. Implementations 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 instruction mechanism specified in *18.2 Extension Instructions*.

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.

Note:

Throughout this specification, an element or attribute that is in no namespace, or an expanded-QName whose namespace part is an empty sequence, is referred to as having a **null namespace URI**.

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. Names of types defined in XML Schema however, are regarded as single words and are capitalized exactly as in XML Schema. This sometimes leads to composite function names such as `current-dateTime`[FO].

3.2 Reserved Namespaces

[DEFINITION: The XSLT namespace, together with certain other namespaces recognized by an XSLT processor, are classified as **reserved namespaces** and MUST be used only as specified in this and related specifications.] The reserved namespaces are those listed below.

- The XSLT namespace, described in *3.1 XSLT Namespace*, is reserved.
- [DEFINITION: The **standard function namespace** `http://www.w3.org/2005/xpath-functions` is used for functions in the function library defined in [Functions and Operators] and standard functions defined in this specification.]

- [DEFINITION: The **XML namespace**, defined in [Namespaces in XML 1.0] as `http://www.w3.org/XML/1998/namespace`, is used for attributes such as `xml:lang`, `xml:space`, and `xml:id`.]
- [DEFINITION: The **schema namespace** `http://www.w3.org/2001/XMLSchema` is used as defined in [XML Schema Part 1]]. In a stylesheet this namespace may be used to refer to built-in schema datatypes and to the constructor functions associated with those datatypes.
- [DEFINITION: The **schema instance namespace** `http://www.w3.org/2001/XMLSchema-instance` is used as defined in [XML Schema Part 1]]. Attributes in this namespace, if they appear in a stylesheet, are treated by the XSLT processor in the same way as any other attributes.

Reserved namespaces may be used without restriction to refer to the names of elements and attributes in source documents and result documents. As far as the XSLT processor is concerned, reserved namespaces other than the XSLT namespace may be used without restriction in the names of literal result elements and user-defined data elements, and in the names of attributes of literal result elements or of XSLT elements: but other processors MAY impose restrictions or attach special meaning to them. Reserved namespaces MUST NOT be used, however, in the names of stylesheet-defined objects such as variables and stylesheet functions.

Note:

With the exception of the XML namespace, any of the above namespaces that are used in a stylesheet must be explicitly declared with a namespace declaration. Although conventional prefixes are used for these namespaces in this specification, any prefix may be used in a user stylesheet.

[ERR XTSE0080] It is a static error to use a reserved namespace in the name of a named template, a mode, an attribute set, a key, a decimal-format, a variable or parameter, a stylesheet function, a named output definition, or a character map.

3.3 Extension Attributes

[DEFINITION: An element from the XSLT namespace may have any attribute not from the XSLT namespace, provided that the expanded-QName (see [XPath 2.0]) of the attribute has a non-null namespace URI. These attributes are referred to as **extension attributes**.] The presence of an extension attribute MUST NOT cause the final result trees produced by the transformation to be different from the result trees that a conformant XSLT 2.0 processor might produce. They MUST NOT cause the processor to fail to signal an error that a conformant processor is required to signal. This means that an extension attribute MUST NOT change the effect of any instruction except to the extent that the effect is implementation-defined or implementation-dependent.

Furthermore, if serialization is performed using one of the serialization methods `xml`, `xhtml`, `html`, or `text` described in *20 Serialization*, the presence of an extension attribute must not cause the serializer to behave in a way that is inconsistent with the mandatory provisions of that specification.

Note:

Extension attributes may be used to modify the behavior of extension functions and extension instructions. They may be used to select processing options in cases where the specification leaves the behavior implementation-defined or implementation-dependent. They may also be used for optimization hints, for diagnostics, or for documentation.

Extension attributes MAY also be used to influence the behavior of the serialization methods `xml`, `xhtml`, `html`, or `text`, to the extent that the behavior of the serialization method is implementation-defined or implementation-dependent. For example, an extension attribute might be used to define the amount of indentation to be used when `indent="yes"` is specified. If a serialization method other than one of these four is requested (using a prefixed QName in the method parameter) then extension attributes may influence its behavior in arbitrary ways. Extension attributes MUST NOT be used to cause the four standard serialization methods to behave in a non-conformant way, for example by failing to report serialization errors that a serializer is REQUIRED to report. An implementation that wishes to provide such options must create a new serialization method for the purpose.

An implementation that does not recognize the name of an extension attribute, or that does not recognize its value, MUST perform the transformation as if the extension attribute were not present. As always, it is permissible to produce warning messages.

The namespace used for an extension attribute will be copied to the result tree in the normal way if it is in scope for a literal result element. This can be prevented using the `[xsl:]exclude-result-prefixes` attribute.

Example: An Extension Attribute for `xsl:message`

The following code might be used to indicate to a particular implementation that the `xsl:message` instruction is to ask the user for confirmation before continuing with the transformation:

```
<xsl:message
    abc:pause="yes"
    xmlns:abc="http://vendor.example.com/xslt/extensions">Phase 1 complete</xsl:message>
```

Implementations that do not recognize the namespace `http://vendor.example.com/xslt/extensions` will simply ignore the extra attribute, and evaluate the `xsl:message` instruction in the normal way.

[ERR XTSE0090] It is a static error for an element from the XSLT namespace to have an attribute whose namespace is either null (that is, an attribute with an unprefixed name) or the XSLT namespace, other than attributes defined for the element in this document.

3.4 XSLT Media Type

The media type `application/xslt+xml` will be registered for XSLT stylesheet modules.

The proposed definition of the media type is at *B The XSLT Media Type*

This media type SHOULD be used for an XML document containing a standard stylesheet module at its top level, and it MAY also be used for a simplified stylesheet module. It SHOULD NOT be used for an XML document containing an embedded stylesheet module.

3.5 Standard Attributes

[DEFINITION: There are a number of **standard attributes** that may appear on any XSLT element: specifically `version`, `exclude-result-prefixes`, `extension-element-prefixes`, `xpath-default-namespace`, `default-collation`, and `use-when`.]

These attributes may also appear on a literal result element, but in this case, to distinguish them from user-defined attributes, the names of the attributes are in the XSLT namespace. They are thus typically written as `xsl:version`, `xsl:exclude-result-prefixes`, `xsl:extension-element-prefixes`, `xsl:xpath-default-namespace`, `xsl:default-collation`, or `xsl:use-when`.

It is RECOMMENDED that all these attributes should also be permitted on extension instructions, but this is at the discretion of the implementer of each extension instruction. They MAY also be permitted on user-defined data elements, though they will only have any useful effect in the case of data elements that are designed to behave like XSLT declarations or instructions.

In the following descriptions, these attributes are referred to generically as `[xsl:]version`, and so on.

These attributes all affect the element they appear on, together with any elements and attributes that have that element as an ancestor. The two forms with and without the XSLT namespace have the same effect; the XSLT namespace is used for the attribute if and only if its parent element is *not* in the XSLT namespace.

In the case of `[xsl:]version`, `[xsl:]xpath-default-namespace`, and `[xsl:]default-collation`, the value can be overridden by a different value for the same attribute appearing on a descendant element. The effective value of the attribute for a particular stylesheet element is determined by the innermost ancestor-or-self element on which the attribute appears.

In an embedded stylesheet module, standard attributes appearing on ancestors of the outermost element of the stylesheet module have no effect.

In the case of `[xsl:]exclude-result-prefixes` and `[xsl:]extension-element-prefixes` the values are cumulative. For these attributes, the value is given as a whitespace-separated list of namespace prefixes, and the effective value for an element is the combined set of namespace URIs designated by the prefixes that appear in this attribute for that element and any of its ancestor elements. Again, the two forms with and without the XSLT namespace are equivalent.

The effect of the `[xsl:]use-when` attribute is described in *3.12 Conditional Element Inclusion*.

Because these attributes may appear on any XSLT element, they are not listed in the syntax summary of each individual element. Instead they are listed and described in the entry for the `xsl:stylesheet` and `xsl:transform` elements only. This reflects the fact that these attributes are often used on the `xsl:stylesheet` element only, in which case they apply to the entire stylesheet module.

Note that the effect of these attributes does *not* extend to stylesheet modules referenced by `xsl:include` or `xsl:import` declarations.

For the detailed effect of each attribute, see the following sections:

`[xsl:]version`

see *3.8 Backwards-Compatible Processing* and *3.9 Forwards-Compatible Processing*

`[xsl:]xpath-default-namespace`

see *5.2 Unprefixed QNames in Expressions and Patterns*

`[xsl:]exclude-result-prefixes`

see *11.1.3 Namespace Nodes for Literal Result Elements*

`[xsl:]extension-element-prefixes`

see *18.2 Extension Instructions*

`[xsl:]use-when`

see *3.12 Conditional Element Inclusion*

`[xsl:]default-collation`

see *3.6.1 The default-collation attribute*

3.6 Stylesheet Element

```
<xsl:stylesheet
  id? = id
  extension-element-prefixes? = tokens
  exclude-result-prefixes? = tokens
  version = number
  xpath-default-namespace? = uri
  default-validation? = "preserve" | "strip"
  default-collation? = uri-list
  input-type-annotations? = "preserve" | "strip" | "unspecified">
  <!-- Content: (xsl:import*, other-declarations) -->
</xsl:stylesheet>
```

```
<xsl:transform
  id? = id
  extension-element-prefixes? = tokens
  exclude-result-prefixes? = tokens
  version = number
  xpath-default-namespace? = uri
  default-validation? = "preserve" | "strip"
  default-collation? = uri-list
```

```
  input-type-annotations? = "preserve" | "strip" | "unspecified">
  <!-- Content: (xsl:import*, other-declarations) -->
</xsl:transform>
```

A stylesheet module is represented by an `xsl:stylesheet` element in an XML document. `xsl:transform` is allowed as a synonym for `xsl:stylesheet`; everything this specification says about the `xsl:stylesheet` element applies equally to `xsl:transform`.

An `xsl:stylesheet` element MUST have a `version` attribute, indicating the version of XSLT that the stylesheet module requires.

[ERR XTSE0110] The value of the `version` attribute MUST be a number: specifically, it MUST be a a valid instance of the type `xs:decimal` as defined in [XML Schema Part 2]. For this version of XSLT, the value SHOULD normally be `2.0`. A value of `1.0` indicates that the stylesheet module was written with the intention that it SHOULD be processed using an XSLT 1.0 processor.

If a stylesheet that specifies `[xsl:]version="1.0"` in the outermost element of the principal stylesheet module (that is, `version="1.0"` in the case of a standard stylesheet module, or `xsl:version="1.0"` in the case of a simplified stylesheet module) is submitted to an XSLT 2.0 processor, the processor SHOULD output a warning advising the user of possible incompatibilities, unless the user has requested otherwise. The processor MUST then process the stylesheet using the rules for backwards-compatible behavior. These rules require that if the processor does not support backwards-compatible behavior, it MUST signal an error and MUST NOT execute the transformation.

When the value of the `version` attribute is greater than 2.0, forwards-compatible behavior is enabled (see *3.9 Forwards-Compatible Processing*).

Note:

XSLT 1.0 allowed the `[xsl:]version` attribute to take any numeric value, and specified that if the value was not equal to 1.0, the stylesheet would be executed in forwards compatible mode. XSLT 2.0 continues to allow the attribute to take any unsigned decimal value. A software product that includes both an XSLT 1.0 processor and an XSLT 2.0 processor (or that can execute as either) may use the `[xsl:]version` attribute to decide which processor to invoke; such behavior is outside the scope of this specification. When the stylesheet is executed with an XSLT 2.0 processor, the value `1.0` is taken to indicate that the stylesheet module was written with XSLT 1.0 in mind: if this value appears on the outermost element of the principal stylesheet module then an XSLT 2.0 processor will either reject the stylesheet or execute it in backwards compatible mode, as described above. Setting `version="2.0"` indicates that the stylesheet is to be executed with neither backwards nor forwards compatible behavior enabled. Any other value less than `2.0` enables backwards compatible behavior, while any value greater than `2.0` enables forwards compatible behavior.

When developing a stylesheet that is designed to execute under either XSLT 1.0 or XSLT 2.0, the recommended practice is to create two alternative stylesheet modules, one specifying `version="1.0"`, and the other specifying `version="2.0"`; these modules can use `xsl:include` or `xsl:import` to incorporate the common code. When running under an XSLT 1.0 processor, the `version="1.0"` module can be selected as the principal stylesheet module; when running under an XSLT 2.0 processor, the `version="2.0"` module can be selected as the principal stylesheet module. Stylesheet modules that are included or imported should specify `version="2.0"` if they make use of XSLT 2.0 facilities, and `version="1.0"` otherwise.

The effect of the `input-type-annotations` attribute is described in *4.3 Stripping Type Annotations from a Source Tree*.

The `default-validation` attribute defines the default value of the `validation` attribute of all `xsl:document`, `xsl:element`, `xsl:attribute`, `xsl:copy`, `xsl:copy-of`, and `xsl:result-document` instructions, and of the `xsl:validation` attribute of all literal result elements. It also determines the validation applied to the implicit final result tree created in the absence of an `xsl:result-document` instruction. This default applies within the stylesheet module: it does not extend to included or imported stylesheet modules. If the attribute is omitted, the default is `strip`. The permitted values are `preserve` and `strip`. For details of the effect of this attribute, see *19.2 Validation*.

[ERR XTSE0120] An `xsl:stylesheet` element MUST NOT have any text node children. (This rule applies after stripping of whitespace text nodes as described in *4.2 Stripping Whitespace from the Stylesheet*.)

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

[DEFINITION: Top-level elements fall into two categories: declarations, and user-defined data elements. Top-level elements whose names are in the XSLT namespace are **declarations**. Top-level elements in any other namespace are user-defined data elements (see *3.6.2 User-defined Data Elements*)].

The declaration elements permitted in the `xsl:stylesheet` element are:

- `xsl:import`
- `xsl:include`
- `xsl:attribute-set`
- `xsl:character-map`
- `xsl:decimal-format`
- `xsl:function`
- `xsl:import-schema`
- `xsl:key`
- `xsl:namespace-alias`
- `xsl:output`
- `xsl:param`
- `xsl:preserve-space`
- `xsl:strip-space`
- `xsl:template`
- `xsl:variable`

Note that the `xsl:variable` and `xsl:param` elements can act either as declarations or as instructions. A global variable or parameter is defined using a declaration; a local variable or parameter using an instruction.

If there are `xsl:import` elements, these MUST come before any other elements. Apart from this, the child elements of the `xsl:stylesheet` element may appear in any order. The ordering of these elements does not affect the results of the transformation unless there are conflicting declarations (for example, two template rules with the same priority that match the same node). In general, it is an error for a stylesheet to contain such conflicting declarations, but in some cases the processor is allowed to recover from the error by choosing the declaration that appears last in the stylesheet.

3.6.1 The `default-collation` attribute

The `default-collation` attribute is a standard attribute that may appear on any element in the XSLT namespace, or (as `xsl:default-collation`) on a literal result element.

The attribute is used to specify the default collation used by all XPath expressions appearing in the attributes of this element, or attributes of descendant elements, unless overridden by another `default-collation` attribute on an inner element. It also determines the collation used by certain XSLT constructs (such as `xsl:key` and `xsl:for-each-group`) within its scope.

The value of the attribute is a whitespace-separated list of collation URIs. If any of these URIs is a relative URI, then it is resolved relative to the base URI of the attribute's parent element. If the implementation recognizes one or more of the resulting absolute collation URIs, then it uses the first one that it recognizes as the default collation.

[ERR XTSE0125] It is a static error if the value of an `[xsl:]default-collation` attribute, after resolving against the base URI, contains no URI that the implementation recognizes as a collation URI.

> **Note:**
>
> The reason the attribute allows a list of collation URIs is that collation URIs will often be meaningful only to one particular XSLT implementation. Stylesheets designed to run with several different implementations can therefore specify several different collation URIs, one for use with each. To avoid the above error condition, it is possible to specify the Unicode Codepoint Collation as the last collation URI in the list.

The `[xsl:]default-collation` attribute does not affect the collation used by `xsl:sort`.

3.6.2 User-defined Data Elements

[DEFINITION: In addition to declarations, the `xsl:stylesheet` element may contain any element not from the XSLT namespace, provided that the expanded-QName of the element has a non-null namespace URI. Such elements are referred to as **user-defined data elements**.]

[ERR XTSE0130] It is a static error if the `xsl:stylesheet` element has a child element whose name has a null namespace URI.

An implementation MAY attach an implementation-defined meaning to user-defined data elements that appear in particular namespaces. The set of namespaces that are recognized for such data elements is implementation-defined. The presence of a user-defined data element MUST NOT change the behavior of XSLT elements and functions defined in this document; for example, it is not permitted for a user-defined data element to specify that `xsl:apply-templates` should use different rules to resolve conflicts. The constraints on what user-defined data elements can and cannot do are exactly the same as the constraints on extension attributes, described in *3.3 Extension Attributes*. Thus, an implementation is always free to ignore user-defined data elements, and MUST ignore such data elements without giving an error if it does not recognize the namespace URI.

User-defined data elements can provide, for example,

- information used by extension instructions or extension functions (see *18 Extensibility and Fallback*),
- information about what to do with any final result tree,
- information about how to construct source trees,
- optimization hints for the processor,
- metadata about the stylesheet,
- structured documentation for the stylesheet.

A user-defined data element MUST NOT precede an `xsl:import` element within a stylesheet module [see ERR XTSE0200]

3.7 Simplified Stylesheet Modules

A simplified syntax is allowed for a stylesheet module that defines only a single template rule for the document node. The stylesheet module may consist of just a literal result element (see *11.1 Literal Result Elements*) together with its contents. The literal result element must have an `xsl:version` attribute (and it must therefore also declare the XSLT namespace). Such a stylesheet module is equivalent to a standard stylesheet module whose `xsl:stylesheet` element contains a template rule containing the literal result element, minus its `xsl:version` attribute; the template rule has a match pattern of `/`.

Example: A Simplified Stylesheet

For example:

```
<html xsl:version="2.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xmlns="http://www.w3.org/1999/xhtml">
  <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="2.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns="http://www.w3.org/1999/xhtml">
<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>
```

Note that it is not possible, using a simplified stylesheet, to request that the serialized output contains a `DOCTYPE` declaration. This can only be done by using a standard stylesheet module, and using the `xsl:output` element.

More formally, a simplified stylesheet module is equivalent to the standard stylesheet module that would be generated by applying the following transformation to the simplified stylesheet module, invoking the transformation by calling the named template `expand`, with the containing literal result element as the context node:

```
<xsl:stylesheet version="2.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template name="expand">
  <xsl:element name="xsl:stylesheet">
    <xsl:attribute name="version" select="@xsl:version"/>
    <xsl:element name="xsl:template">
      <xsl:attribute name="match">/</xsl:attribute>
      <xsl:copy-of select="."/>
    </xsl:element>
  </xsl:element>
</xsl:template>

</xsl:stylesheet>
```

[ERR XTSE0150] A literal result element that is used as the outermost element of a simplified stylesheet module MUST have an `xsl:version` attribute. This indicates the version of XSLT that the stylesheet requires. For this version of XSLT, the value will normally be `2.0`; the value MUST be a valid instance of the type `xs:decimal` as defined in [XML Schema Part 2].

Other literal result elements may also have an `xsl:version` attribute. When the `xsl:version` attribute is numerically less than `2.0`, backwards-compatible processing behavior is enabled (see *3.8 Backwards-Compatible Processing*). When the `xsl:version` attribute is numerically greater than `2.0`, forwards-compatible behavior is enabled (see *3.9 Forwards-Compatible Processing*).

The allowed content of a literal result element when used as a simplified stylesheet is the same as when it occurs within a sequence constructor. Thus, a literal result element used as the document element of a simplified stylesheet cannot contain declarations. Simplified stylesheets therefore cannot use global variables, stylesheet parameters, stylesheet functions, keys, attribute-sets, or output definitions. In turn this means that the only useful way to initiate the transformation is to supply a document node as the initial context node, to be matched by the implicit `match="/"` template rule using the default mode.

3.8 Backwards-Compatible Processing

[DEFINITION: An element enables backwards-compatible behavior for itself, its attributes, its descendants and their attributes if it has an `[xsl:]version` attribute (see *3.5 Standard Attributes*) whose value is less than `2.0`.]

An element that has an `[xsl:]version` attribute whose value is greater than or equal to `2.0` disables backwards-compatible behavior for itself, its attributes, its descendants and their attributes. The compatibility behavior established by an element overrides any compatibility behavior established by an ancestor element.

If an attribute containing an XPath expression is processed with backwards-compatible behavior, then the expression is evaluated with XPath 1.0 compatibility mode set to `true`. For details of this mode, see Section 2.1.1 Static Context[XP]. Furthermore, in such an expression any function call for which no implementation is available (unless it uses the standard function namespace) is bound to a fallback error function whose effect when evaluated is to raise a dynamic error [see ERR XTDE1425] . The effect is that with backwards-compatible behavior enabled, calls on extension functions that are not available in a particular implementation do not cause an error unless the function call is actually evaluated. For further details, see *18.1 Extension Functions*.

Note:

This might appear to contradict the specification of XPath 2.0, which states that a static error [XPST0017] is raised when an expression contains a call to a function that is not present (with matching name and arity) in the static context. This apparent contradiction is resolved by specifying that the XSLT processor constructs a static context for the expression in which every possible function name and arity (other than names in the standard function namespace) is present; when no other implementation of the function is available, the function call is bound to a fallback error function whose run-time effect is to raise a dynamic error.

Certain XSLT constructs also produce different results when backwards-compatible behavior is enabled. This is described separately for each such construct.

These rules do not apply to the `xsl:output` element, whose `version` attribute has an entirely different purpose: it is used to define the version of the output method to be used for serialization.

Note:

By making use of backwards-compatible behavior, it is possible to write the stylesheet in a way that ensures that its results when processed with an XSLT 2.0 processor are identical to the effects of processing the same stylesheet using an XSLT 1.0 processor. The differences are described (non-normatively) in *J.1 Incompatible Changes*. To assist with transition, some parts of a stylesheet may be processed with backwards compatible behavior enabled, and other parts with this behavior disabled. All data values manipulated by an XSLT 2.0 processor are defined by the XDM data model, whether or not the relevant expressions use backwards compatible behavior. Because the same data model is used in both cases, expressions are fully composable. The result of evaluating instructions or expressions with backwards compatible behavior is fully defined in the XSLT 2.0 and XPath 2.0 specifications, it is not defined by reference to the XSLT 1.0 and XPath 1.0 specifications.

It is implementation-defined whether a particular XSLT 2.0 implementation supports backwards-compatible behavior.

[ERR XTDE0160] If an implementation does not support backwards-compatible behavior, then it is a non-recoverable dynamic error if any element is evaluated that enables backwards-compatible behavior.

Note:

To write a stylesheet that works with both XSLT 1.0 and 2.0 processors, while making selective use of XSLT 2.0 facilities, it is necessary to understand both the rules for backwards-compatible behavior in XSLT 2.0, and the rules for forwards-compatible behavior in XSLT 1.0. If the `xsl:stylesheet` element specifies `version="2.0"`, then an XSLT 1.0 processor will ignore XSLT 2.0 declarations that were not defined in XSLT 1.0, for example `xsl:function` and `xsl:import-schema`. If any new XSLT 2.0 instructions are used (for example `xsl:analyze-string` or `xsl:namespace`), or if new XPath 2.0 features are used (for example, new functions, or syntax such as conditional expressions, or calls to a function defined using `xsl:function`), then the stylesheet must provide fallback behavior that relies on XSLT 1.0 and XPath 1.0 facilities only. The fallback behavior can be invoked by using the `xsl:fallback` instruction, or by testing the results of the `function-available` or `element-available` functions, or by testing the value of the `xsl:version` property returned by the `system-property` function.

3.9 Forwards-Compatible Processing

The intent of forwards-compatible behavior is to make it possible to write a stylesheet that takes advantage of features introduced in some version of XSLT subsequent to XSLT 2.0, while retaining the ability to execute the stylesheet with an XSLT 2.0 processor using appropriate fallback behavior.

It is always possible to write conditional code to run under different XSLT versions by using the `use-when` feature described in *3.12 Conditional Element Inclusion*. The rules for forwards-compatible behavior supplement this mechanism in two ways:

- certain constructs in the stylesheet that mean nothing to an XSLT 2.0 processor are ignored, rather than being treated as errors.
- explicit fallback behavior can be defined for instructions defined in a future XSLT release, using the `xsl:fallback` instruction.

The detailed rules follow.

[DEFINITION: An element enables **forwards-compatible behavior** for itself, its attributes, its descendants and their attributes if it has an `[xsl:]version` attribute (see *3.5 Standard Attributes*) whose value is greater than `2.0`.]

An element that has an `[xsl:]version` attribute whose value is less than or equal to `2.0` disables forwards-compatible behavior for itself, its attributes, its descendants and their attributes. The compatibility behavior established by an element overrides any compatibility behavior established by an ancestor element.

These rules do not apply to the `version` attribute of the `xsl:output` element, which has an entirely different purpose: it is used to define the version of the output method to be used for serialization.

Within a section of a stylesheet where forwards-compatible behavior is enabled:

- if an element in the XSLT namespace appears as a child of the `xsl:stylesheet` element, and XSLT 2.0 does not allow such elements to occur as children of the `xsl:stylesheet` element, then the element and its content MUST be ignored.
- if an element has an attribute that XSLT 2.0 does not allow the element to have, then the attribute MUST be ignored.
- if an element in the XSLT namespace appears as part of a sequence constructor, and XSLT 2.0 does not allow such elements to appear as part of a sequence constructor, then:
  1. If the element has one or more `xsl:fallback` children, then no error is reported either statically or dynamically, and the result of evaluating the instruction is the concatenation of the sequences formed by evaluating the sequence constructors within its `xsl:fallback` children, in document order. Siblings of the `xsl:fallback` elements are ignored, even if they are valid XSLT 2.0 instructions.
  2. If the element has no `xsl:fallback` children, then a static error is reported in the same way as if forwards-compatible behavior were not enabled.

Example: Forwards Compatible Behavior

For example, an XSLT 2.0 processor will 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="17.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <xsl:exciting-new-17.0-feature>
      <xsl:fly-to-the-moon/>
      <xsl:fallback>
        <html>
          <head>
            <title>XSLT 17.0 required</title>
          </head>
          <body>
            <p>Sorry, this stylesheet requires XSLT 17.0.</p>
          </body>
        </html>
      </xsl:fallback>
    </xsl:exciting-new-17.0-feature>
  </xsl:template>
</xsl:stylesheet>
```

Note:

If a stylesheet depends crucially on a declaration introduced by a version of XSLT after 2.0, then the stylesheet can use an `xsl:message` element with `terminate="yes"` (see *17 Messages*) to ensure that implementations that conform to an earlier version of XSLT will not silently ignore the declaration.

Example: Testing the XSLT Version

For example,

```
<xsl:stylesheet version="18.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:important-new-17.0-declaration/>

  <xsl:template match="/">
    <xsl:choose>
      <xsl:when test="number(system-property('xsl:version')) lt 17.0">
        <xsl:message terminate="yes">
          <xsl:text>Sorry, this stylesheet requires XSLT 17.0.</xsl:text>
        </xsl:message>
      </xsl:when>
      <xsl:otherwise>
        ...
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>
  ...
</xsl:stylesheet>
```

3.10 Combining Stylesheet Modules

XSLT provides two mechanisms to construct a stylesheet from multiple stylesheet modules:

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

3.10.1 Locating Stylesheet Modules

The include and import mechanisms use two declarations, `xsl:include` and `xsl:import`, which are defined in the sections that follow.

These declarations use an `href` attribute, whose value is a URI reference, to identify the stylesheet module to be included or imported. If the value of this attribute is a relative URI, it is resolved as described in *5.8 URI References*.

After resolving against the base URI, the way in which the URI reference is used to locate a representation of a stylesheet module, and the way in which the stylesheet module is constructed from that representation, are implementation-defined. In particular, it is implementation-defined which URI schemes are supported, whether fragment identifiers are supported, and what media types are supported. Conventionally, the URI is a reference to a resource containing the stylesheet module as a source XML document, or it may include a fragment identifier that selects an embedded stylesheet module within a source XML document; but the implementation is free to use other mechanisms to locate the stylesheet module identified by the URI reference.

The referenced stylesheet module may be any of the four kinds of stylesheet module: that is, it may be standalone or embedded, and it may be standard or simplified. If it is a simplified stylesheet module then it is transformed into the equivalent standard stylesheet module by applying the transformation described in *3.7 Simplified Stylesheet Modules*.

Implementations MAY choose to accept URI references containing a fragment identifier defined by reference to the XPointer specification (see [XPointer Framework]). Note that if the implementation does not support the use of fragment identifiers in the URI reference, then it will not be possible to include an embedded stylesheet module.

[ERR XTSE0165] It is a static error if the processor is not able to retrieve the resource identified by the URI reference, or if the resource that is retrieved does not contain a stylesheet module conforming to this specification.

3.10.2 Stylesheet Inclusion

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

A stylesheet module may include another stylesheet module using an `xsl:include` declaration.

The `xsl:include` declaration has a REQUIRED `href` attribute whose value is a URI reference identifying the stylesheet module to be included. This attribute is used as described in *3.10.1 Locating Stylesheet Modules*.

[ERR XTSE0170] An `xsl:include` element MUST be a top-level element.

[DEFINITION: A **stylesheet level** is a collection of stylesheet modules connected using `xsl:include` declarations: specifically, two stylesheet modules *A* and *B* are part of the same stylesheet level if one of them includes the other by means of an `xsl:include` declaration, or if there is a third stylesheet module *C* that is in the same stylesheet level as both *A* and *B*.]

[DEFINITION: The declarations within a stylesheet level have a total ordering known as **declaration order**. The order of declarations within a stylesheet level is the same as the document order that would result if each stylesheet module were inserted textually in place of the `xsl:include` element that references it.] In other respects, however, the effect of `xsl:include` is not equivalent to the effect that would be obtained by textual inclusion.

[ERR XTSE0180] It is a static error if a stylesheet module directly or indirectly includes itself.

Note:

It is not intrinsically an error for a stylesheet to include the same module more than once. However, doing so 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'*.

3.10.3 Stylesheet Import

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

A stylesheet module may import another stylesheet module using an `xsl:import` declaration. Importing a stylesheet module is the same as including it (see *3.10.2 Stylesheet Inclusion*) except that template rules and other declarations in the importing module take precedence over template rules and declarations in the imported module; this is described in more detail below.

The `xsl:import` declaration has a REQUIRED `href` attribute whose value is a URI reference identifying the stylesheet module to be included. This attribute is used as described in *3.10.1 Locating Stylesheet Modules*.

[ERR XTSE0190] An `xsl:import` element MUST be a top-level element.

[ERR XTSE0200] The `xsl:import` element children MUST precede all other element children of an `xsl:stylesheet` element, including any `xsl:include` element children and any user-defined data elements.

Example: Using `xsl:import`

For example,

```
<xsl:stylesheet version="2.0"
                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>
```

[DEFINITION: The stylesheet levels making up a stylesheet are treated as forming an **import tree**. In the import tree, each stylesheet level has one child for each `xsl:import` declaration that it contains.] The ordering of the children is the declaration order of the `xsl:import` declarations within their stylesheet level.

[DEFINITION: A declaration *D* in the stylesheet is defined to have lower **import precedence** than another declaration *E* if the stylesheet level containing *D* would be visited before the stylesheet level containing *E* in a post-order traversal of the import tree (that is, a traversal of the import tree in which a stylesheet level is visited after its children). Two declarations within the same stylesheet level have the same import precedence.]

For example, suppose

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

Then the import tree has the following structure:

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

In general, a declaration with higher import precedence takes precedence over a declaration with lower import precedence. This is defined in detail for each kind of declaration.

[ERR XTSE0210] It is a static error if a stylesheet module directly or indirectly imports itself.

Note:

The case where a stylesheet module with a particular URI is imported several times is not treated specially. The effect is exactly the same as if several stylesheet modules with different URIs but identical content were imported. This might or might not cause an error, depending on the content of the stylesheet module.

3.11 Embedded Stylesheet Modules

An embedded stylesheet module is a stylesheet module whose containing element is not the outermost element of the containing XML document. Both standard stylesheet modules and simplified stylesheet modules may be embedded in this way.

Two situations where embedded stylesheets may be useful are:

- The stylesheet may be embedded in the source document to be transformed.
- The stylesheet may be embedded in an XML document that describes a sequence of processing of which the XSLT transformation

forms just one part.

The xsl:stylesheet element MAY have an id attribute to facilitate reference to the stylesheet module within the containing document.

Note:

In order for such an attribute value to be used as a fragment identifier in a URI, the XDM attribute node must generally have the is-id property: see Section 5.5 is-id Accessor[DM]. This property will typically be set if the attribute is defined in a DTD as being of type ID, or if is defined in a schema as being of type xs:ID. It is also necessary that the media type of the containing document should support the use of ID values as fragment identifiers. Such support is widespread in existing products, and is expected to be endorsed in respect of the media type application/xml by a future revision of [RFC3023].

An alternative, if the implementation supports it, is to use an xml:id attribute. XSLT allows this attribute (like other namespaced attributes) to appear on any XSLT element.

Example: The xml-stylesheet Processing Instruction

The following example shows how the xml-stylesheet processing instruction (see [XML Stylesheet]) can be used to allow a source 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="application/xslt+xml" href="#style1"?>
<!DOCTYPE doc SYSTEM "doc.dtd">
<doc>
<head>
<xsl:stylesheet id="style1"
                version="2.0"
                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 module that is embedded in the document to which it is to be applied typically needs to contain a template rule that specifies that xsl:stylesheet elements are to be ignored.

Note:

The above example uses the pseudo-attribute type="application/xslt+xml" in the xml-stylesheet processing instruction to denote an XSLT stylesheet. This usage is subject to confirmation: see *3.4 XSLT Media Type*. In the absence of a registered media type for XSLT stylesheets, some vendors' products have adopted different conventions, notably type="text/xsl".

Note:

Support for the xml-stylesheet processing instruction is not required for conformance with this Recommendation. Implementations are not constrained in the mechanisms they use to identify a stylesheet when a transformation is initiated: see *2.3 Initiating a Transformation*.

3.12 Conditional Element Inclusion

Any element in the XSLT namespace may have a use-when attribute whose value is an XPath expression that can be evaluated statically. If the attribute is present and the effective boolean value[XP] of the expression is false, then the element, together with all the nodes having that element as an ancestor, is effectively excluded from the stylesheet module. When a node is effectively excluded from a stylesheet module the stylesheet module has the same effect as if the node were not there. Among other things this means that no static or dynamic errors will be reported in respect of the element and its contents, other than errors in the use-when attribute itself.

Note:

This does not apply to XML parsing or validation errors, which will be reported in the usual way. It also does not apply to attributes that are necessarily processed before [xsl:]use-when, examples being xml:space and [xsl:]xpath-default-namespace.

A literal result element, or any other element within a stylesheet module that is not in the XSLT namespace, may similarly carry an xsl:use-when attribute.

If the xsl:stylesheet or xsl:transform element itself is effectively excluded, the effect is to exclude all the children of the xsl:stylesheet or xsl:transform element, but not the xsl:stylesheet or xsl:transform element or its attributes.

Note:

This allows all the declarations that depend on the same condition to be included in one stylesheet module, and for their inclusion or exclusion to be controlled by a single use-when attribute at the level of the module.

Conditional element exclusion happens after stripping of whitespace text nodes from the stylesheet, as described in *4.2 Stripping Whitespace from the Stylesheet*.

There are no syntactic constraints on the XPath expression that can be used as the value of the `use-when` attribute. However, there are severe constraints on the information provided in its evaluation context. These constraints are designed to ensure that the expression can be evaluated at the earliest possible stage of stylesheet processing, without any dependency on information contained in the stylesheet itself or in any source document.

Specifically, the components of the static and dynamic context are defined by the following two tables:

Static Context Components for `use-when` Expressions

Component	Value
XPath 1.0 compatibility mode	false
In scope namespaces	determined by the in-scope namespaces for the containing element in the stylesheet
Default element/type namespace	determined by the `xpath-default-namespace` attribute if present (see *5.2 Unprefixed QNames in Expressions and Patterns*); otherwise the null namespace
Default function namespace	The standard function namespace
In scope type definitions	The type definitions that would be available in the absence of any `xsl:import-schema` declaration
In scope element declarations	None
In scope attribute declarations	None
In scope variables	None
In scope functions	The core functions defined in [Functions and Operators], together with the functions `element-available`, `function-available`, `type-available`, and `system-property` defined in this specification, plus the set of extension functions that are present in the static context of every XPath expression (other than a use-when expression) within the content of the element that is the parent of the `use-when` attribute. Note that stylesheet functions are *not* included in the context, which means that the function `function-available` will return `false` in respect of such functions. The effect of this rule is to ensure that `function-available` returns true in respect of functions that can be called within the scope of the `use-when` attribute. It also has the effect that these extensions functions will be recognized within the `use-when` attribute itself; however, the fact that a function is available in this sense gives no guarantee that a call on the function will succeed.
In scope collations	Implementation-defined
Default collation	The Unicode Codepoint Collation
Base URI	The base URI of the containing element in the stylesheet
Statically known documents	None
Statically known collections	None

Dynamic Context Components for `use-when` Expressions

Component	Value
Context item, position, and size	Undefined
Dynamic variables	None
Current date and time	Implementation-defined
Implicit timezone	Implementation-defined
Available documents	None
Available collections	None

Within a stylesheet module, all expressions contained in `[xsl:]use-when` attributes are evaluated in a single execution scope[FO]. This need not be the same execution scope as that used for `[xsl]:use-when` expressions in other stylesheet modules, or as that used when evaluating XPath expressions appearing elsewhere in the stylesheet module. This means that a function such as `current-date`[FO] will return the same result when called in different `[xsl:]use-when` expressions within the same stylesheet module, but will not necessarily return the same result as the same call in an `[xsl:]use-when` expression within a different stylesheet module, or as a call on the same function executed during the transformation proper.

The use of `[xsl:]use-when` is illustrated in the following examples.

Example: Using Conditional Exclusion to Achieve Portability

This example demonstrates the use of the `use-when` attribute to achieve portability of a stylesheet across schema-aware and non-schema-aware processors.

```
<xsl:import-schema schema-location="http://example.com/schema"
```

```
              use-when="system-property('xsl:is-schema-aware')='yes'"/>

  <xsl:template match="/"
              use-when="system-property('xsl:is-schema-aware')='yes'"
              priority="2">
    <xsl:result-document validation="strict">
      <xsl:apply-templates/>
    </xsl:result-document>
  </xsl:template>

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

The effect of these declarations is that a non-schema-aware processor ignores the `xsl:import-schema` declaration and the first template rule, and therefore generates no errors in respect of the schema-related constructs in these declarations.

Example: Including Variant Stylesheet Modules

This example includes different stylesheet modules depending on which XSLT processor is in use.

```
  <xsl:include href="module-A.xsl"
       use-when="system-property('xsl:vendor')='vendor-A'"/>
  <xsl:include href="module-B.xsl"
       use-when="system-property('xsl:vendor')='vendor-B'"/>
```

3.13 Built-in Types

Every XSLT 2.0 processor includes the following named type definitions in the in-scope schema components:

- All the primitive atomic types defined in [XML Schema Part 2], with the exception of `xs:NOTATION`. That is: `xs:string`, `xs:boolean`, `xs:decimal`, `xs:double`, `xs:float`, `xs:date`, `xs:time`, `xs:dateTime`, `xs:duration`, `xs:QName`, `xs:anyURI`, `xs:gDay`, `xs:gMonthDay`, `xs:gMonth`, `xs:gYearMonth`, `xs:gYear`, `xs:base64Binary`, and `xs:hexBinary`.
- The derived atomic type `xs:integer` defined in [XML Schema Part 2].
- The types `xs:anyType` and `xs:anySimpleType`.
- The following types defined in [XPath 2.0]: `xs:yearMonthDuration`, `xs:dayTimeDuration`, `xs:anyAtomicType`, `xs:untyped`, and `xs:untypedAtomic`.

A schema-aware XSLT processor additionally supports:

- All other built-in types defined in [XML Schema Part 2]
- User-defined types, and element and attribute declarations, that are imported using an `xsl:import-schema` declaration as described in *3.14 Importing Schema Components*. These may include both simple and complex types.

  Note:

  The names that are imported from the XML Schema namespace do not include all the names of top-level types defined in either the Schema for Schemas or the Schema for Datatypes. The Schema for Datatypes, as well as defining built-in types such as `xs:integer` and `xs:double`, also defines types that are intended for use only within the Schema for DataTypes, such as `xs:derivationControl`. A stylesheet that is designed to process XML Schema documents as its input or output may import the Schema for Schemas.

An implementation may define mechanisms that allow additional schema components to be added to the in-scope schema components for the stylesheet. For example, the mechanisms used to define extension functions (see *18.1 Extension Functions*) may also be used to import the types used in the interface to such functions.

These schema components are the only ones that may be referenced in XPath expressions within the stylesheet, or in the `[xsl:]type` and `as` attributes of those elements that permit these attributes.

For a Basic XSLT Processor, schema built-in types that are not included in the static context (for example, `xs:NCName`) are "unknown types" in the sense of Section 2.5.4 SequenceType Matching[XP]. In the language of that section, a Basic XSLT Processor MUST be able to determine whether these unknown types are derived from known schema types such as `xs:string`. The purpose of this rule is to ensure that system functions such as `local-name-from-QName`[FO], which is defined to return an `xs:NCName`, behave correctly. A stylesheet that uses a Basic XSLT Processor will not be able to test whether the returned value is an `xs:NCName`, but it will be able to use it as if it were an `xs:string`.

3.14 Importing Schema Components

Note:

The facilities described in this section are not available with a basic XSLT processor. They require a schema-aware XSLT processor, as described in *21 Conformance*.

```
<!-- Category: declaration -->
<xsl:import-schema
  namespace? = uri-reference
  schema-location? = uri-reference>
  <!-- Content: xs:schema? -->
</xsl:import-schema>
```

The `xsl:import-schema` declaration is used to identify schema components (that is, top-level type definitions and top-level element and

attribute declarations) that need to be available statically, that is, before any source document is available. Names of such components used statically within the stylesheet must refer to an in-scope schema component, which means they must either be built-in types as defined in *3.13 Built-in Types*, or they must be imported using an `xsl:import-schema` declaration.

The `xsl:import-schema` declaration identifies a namespace containing the names of the components to be imported (or indicates that components whose names are in no namespace are to be imported). The effect is that the names of top-level element and attribute declarations and type definitions from this namespace (or non-namespace) become available for use within XPath expressions in the stylesheet, and within other stylesheet constructs such as the `type` and `as` attributes of various XSLT elements.

The same schema components are available in all stylesheet modules; importing components in one stylesheet module makes them available throughout the stylesheet.

The `namespace` and `schema-location` attributes are both optional.

If the `xsl:import-schema` element contains an `xs:schema` element, then the `schema-location` attribute must be absent, and the `namespace` attribute must either have the same value as the `targetNamespace` attribute of the `xs:schema` element (if present), or must be absent, in which case its effective value is that of the `targetNamespace` attribute of the `xs:schema` element if present or the zero-length string otherwise.

[ERR XTSE0215] It is a static error if an `xsl:import-schema` element that contains an `xs:schema` element has a `schema-location` attribute, or if it has a `namespace` attribute that conflicts with the target namespace of the contained schema.

If two `xsl:import-schema` declarations specify the same namespace, or if both specify no namespace, then only the one with highest import precedence is used. If this leaves more than one, then all the declarations at the highest import precedence are used (which may cause conflicts, as described below).

After discarding any `xsl:import-schema` declarations under the above rule, the effect of the remaining `xsl:import-schema` declarations is defined in terms of a hypothetical document called the synthetic schema document, which is constructed as follows. The synthetic schema document defines an arbitrary target namespace that is different from any namespace actually used by the application, and it contains `xs:import` elements corresponding one-for-one with the `xsl:import-schema` declarations in the stylesheet, with the following correspondence:

- The `namespace` attribute of the `xs:import` element is copied from the `namespace` attribute of the `xsl:import-schema` declaration if it is explicitly present, or is implied by the `targetNamespace` attribute of a contained `xs:schema` element, and is absent if it is absent.
- The `schemaLocation` attribute of the `xs:import` element is copied from the `schema-location` attribute of the `xsl:import-schema` declaration if present, and is absent if it is absent. If there is a contained `xs:schema` element, the effective value of the `schemaLocation` attribute is a URI referencing a document containing a copy of the `xs:schema` element.
- The base URI of the `xs:import` element is the same as the base URI of the `xsl:import-schema` declaration.

The schema components included in the in-scope schema components (that is, the components whose names are available for use within the stylesheet) are the top-level element and attribute declarations and type definitions that are available for reference within the synthetic schema document. See [XML Schema Part 1] (section 4.2.3, *References to schema components across namespaces*).

[ERR XTSE0220] It is a static error if the synthetic schema document does not satisfy the constraints described in [XML Schema Part 1] (section 5.1, *Errors in Schema Construction and Structure*). This includes, without loss of generality, conflicts such as multiple definitions of the same name.

Note:

The synthetic schema document does not need to be constructed by a real implementation. It is purely a mechanism for defining the semantics of `xsl:import-schema` in terms of rules that already exist within the XML Schema specification. In particular, it implicitly defines the rules that determine whether the set of `xsl:import-schema` declarations are mutually consistent.

These rules do not cause names to be imported transitively. The fact that a name is available for reference within a schema document A does not of itself make the name available for reference in a stylesheet that imports the target namespace of schema document A. (See [XML Schema Part 1] section 3.15.3, Constraints on XML Representations of Schemas.) The stylesheet must import all the namespaces containing names that it actually references.

The `namespace` attribute indicates that a schema for the given namespace is required by the stylesheet. This information may be enough on its own to enable an implementation to locate the required schema components. The `namespace` attribute may be omitted to indicate that a schema for names in no namespace is being imported. The zero-length string is not a valid namespace URI, and is therefore not a valid value for the `namespace` attribute.

The `schema-location` attribute is a URI Reference that gives a hint indicating where a schema document or other resource containing the required definitions may be found. It is likely that a schema-aware XSLT processor will be able to process a schema document found at this location.

The XML Schema specification gives implementations flexibility in how to handle multiple imports for the same namespace. Multiple imports do not cause errors if the definitions do not conflict.

A consequence of these rules is that it is not intrinsically an error if no schema document can be located for a namespace identified in an `xsl:import-schema` declaration. This will cause an error only if it results in the stylesheet containing references to names that have not been imported.

An inline schema document (using an `xs:schema` element as a child of the `xsl:import-schema` element) has the same status as an external schema document, in the sense that it acts as a hint for a source of schema components in the relevant namespace. To ensure that the inline schema document is always used, it is advisable to use a target namespace that is unique to this schema document.

The use of a namespace in an `xsl:import-schema` declaration does not by itself associate any namespace prefix with the namespace. If names from the namespace are used within the stylesheet module then a namespace declaration must be included in the stylesheet module, in the usual way.

Example: An Inline Schema Document

The following example shows an inline schema document. This declares a simple type `local:yes-no`, which the stylesheet then uses in the declaration of a variable.

The example assumes the namespace declaration `xmlns:local="http://localhost/ns/yes-no"`

```
<xsl:import-schema>
  <xs:schema targetNamespace="http://localhost/ns/yes-no"
             xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:simpleType name="local:yes-no">
      <xs:restriction base="xs:string">
        <xs:enumeration value="yes"/>
        <xs:enumeration value="no"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:schema>
</xsl:import-schema>

<xs:variable name="condition" select="'yes'" as="local:yes-no"/>
```

4 Data Model

The data model used by XSLT is the XPath 2.0 and XQuery 1.0 data model (XDM), as defined in [Data Model]. XSLT operates on source, result and stylesheet documents using the same data model.

This section elaborates on some particular features of XDM as it is used by XSLT:

The rules in *4.2 Stripping Whitespace from the Stylesheet* and *4.4 Stripping Whitespace from a Source Tree* make use of the concept of a whitespace text node.

[DEFINITION: A **whitespace text node** is a text node whose content consists entirely of whitespace characters (that is, #x09, #x0A, #x0D, or #x20).]

Note:

Features of a source XML document that are not represented in the XDM tree will have no effect on the operation of an XSLT stylesheet. Examples of such features are entity references, CDATA sections, character references, whitespace within element tags, and the choice of single or double quotes around attribute values.

4.1 XML Versions

The XDM data model defined in [Data Model] is capable of representing either an XML 1.0 document (conforming to [XML 1.0] and [Namespaces in XML 1.0]) or an XML 1.1 document (conforming to [XML 1.1] and [Namespaces in XML 1.1]), and it makes no distinction between the two. In principle, therefore, XSLT 2.0 can be used with either of these XML versions.

Construction of the XDM tree is outside the scope of this specification, so XSLT 2.0 places no formal requirements on an XSLT processor to accept input from either XML 1.0 documents or XML 1.1 documents or both. This specification does define a serialization capability (see *20 Serialization*), though from a conformance point of view it is an optional feature. Although facilities are described for serializing the XDM tree as either XML 1.0 or XML 1.1 (and controlling the choice), there is again no formal requirement on an XSLT processor to support either or both of these XML versions as serialization targets.

Because the XDM tree is the same whether the original document was XML 1.0 or XML 1.1, the semantics of XSLT processing do not depend on the version of XML used by the original document. There is no reason in principle why all the input and output documents used in a single transformation must conform to the same version of XML.

Some of the syntactic constructs in XSLT 2.0 and XPath 2.0, for example the productions Char [XML] and NCName [Names], are defined by reference to the XML and XML Namespaces specifications. There are slight variations between the XML 1.0 and XML 1.1 versions of these productions. Implementations MAY support either version; it is RECOMMENDED that an XSLT 2.0 processor that implements the 1.1 versions SHOULD also provide a mode that supports the 1.0 versions. It is thus implementation-defined whether the XSLT processor supports XML 1.0 with XML Namespaces 1.0, or XML 1.1 with XML Namespaces 1.1, or supports both versions at user option.

Note:

The specification referenced as [Namespaces in XML 1.0] was actually published without a version number.

At the time of writing there is no published version of [XML Schema Part 2] that references the XML 1.1 specifications. This means that data types such as `xs:NCName` and `xs:ID` are constrained by the XML 1.0 rules, and do not allow the full range of values permitted by XML 1.1. This situation will not be resolved until a new version of [XML Schema Part 2] becomes available; in the meantime, it is RECOMMENDED that implementers wishing to support XML 1.1 should consult [XML Schema 1.0 and XML 1.1] for guidance. An XSLT 2.0 processor that supports XML 1.1 SHOULD implement the rules in later versions of [XML Schema Part 2] as they become available.

4.2 Stripping Whitespace from the Stylesheet

The tree representing the stylesheet is preprocessed as follows:

1. All comments and processing instructions are removed.
2. Any text nodes that are now adjacent to each other are merged.
3. Any whitespace text node that satisfies both the following conditions is removed from the tree:
   - The parent of the text node is not an `xsl:text` element
   - The text node does not have an ancestor element that has an `xml:space` attribute with a value of `preserve`, unless there is a closer ancestor element having an `xml:space` attribute with a value of `default`.
4. Any whitespace text node whose parent is one of the following elements is removed from the tree, regardless of any `xml:space` attributes:

xsl:analyze-string
xsl:apply-imports
xsl:apply-templates
xsl:attribute-set
xsl:call-template
xsl:character-map
xsl:choose
xsl:next-match
xsl:stylesheet
xsl:transform

5. Any whitespace text node whose following-sibling node is an xsl:param or xsl:sort element is removed from the tree, regardless of any xml:space attributes.

[ERR XTSE0260] Within an XSLT element that is REQUIRED to be empty, any content other than comments or processing instructions, including any whitespace text node preserved using the xml:space="preserve" attribute, is a static error.

Note:

Using xml:space="preserve" in parts of the stylesheet that contain sequence constructors will cause all text nodes in that part of the stylesheet, including those that contain whitespace only, to be copied to the result of the sequence constructor. When the result of the sequence constructor is used to form the content of an element, this can cause errors if such text nodes are followed by attribute nodes generated using xsl:attribute.

Note:

If an xml:space attribute is specified on a literal result element, it will be copied to the result tree in the same way as any other attribute.

4.3 Stripping Type Annotations from a Source Tree

[DEFINITION: The term **type annotation** is used in this specification to refer to the value returned by the dm:type-name accessor of a node: see Section 5.14 type-name Accessor[DM].]

There is sometimes a requirement to write stylesheets that produce the same results whether or not the source documents have been validated against a schema. To achieve this, an option is provided to remove any type annotations on element and attribute nodes in a source tree, replacing them with an annotation of xs:untyped in the case of element nodes, and xs:untypedAtomic in the case of attribute nodes.

Such stripping of type annotations can be requested by specifying input-type-annotations="strip" on the xsl:stylesheet element. This attribute has three permitted values: strip, preserve, and unspecified. The default value is unspecified. Stripping of type annotations takes place if at least one stylesheet module in the stylesheet specifies input-type-annotations="strip".

[ERR XTSE0265] It is a static error if there is a stylesheet module in the stylesheet that specifies input-type-annotations="strip" and another stylesheet module that specifies input-type-annotations="preserve".

The source trees to which this applies are the same as those affected by xsl:strip-space and xsl:preserve-space: see *4.4 Stripping Whitespace from a Source Tree*.

When type annotations are stripped, the following changes are made to the source tree:

- The type annotation of every element node is changed to xs:untyped
- The type annotation of every attribute node is changed to xs:untypedAtomic
- The typed value of every element and attribute node is set to be the same as its string value, as an instance of xs:untypedAtomic.
- The is-nilled property of every element node is set to false.

The values of the is-id and is-idrefs properties are not changed.

Note:

Stripping type annotations does not necessarily return the document to the state it would be in had validation not taken place. In particular, any defaulted elements and attributes that were added to the tree by the validation process will still be present , and elements and attributes validated as IDs will still be accessible using the id[FO] function.

4.4 Stripping Whitespace from a Source Tree

A source tree supplied as input to the transformation process may contain whitespace text nodes that are of no interest, and that do not need to be retained by the transformation. Conceptually, an XSLT processor makes a copy of the source tree from which unwanted whitespace text nodes have been removed. This process is referred to as whitespace stripping.

For the purposes of this section, the term **source tree** means the document containing the initial context node, and any document returned by the functions document, doc[FO], or collection[FO]. It does not include documents passed as the values of stylesheet parameters or returned from extension functions.

The stripping process takes as input a set of element names whose child whitespace text nodes are to be preserved. The way in which this set of element names is established using the xsl:strip-space and xsl:preserve-space declarations is described later in this section.

A whitespace text node is preserved if either of the following apply:

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

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

```
<!-- Category: declaration -->
<xsl:strip-space
  elements = tokens />
```

```
<!-- Category: declaration -->
<xsl:preserve-space
  elements = tokens />
```

The set of whitespace-preserving element names is specified by `xsl:strip-space` and `xsl:preserve-space` declarations. Whether an element name is included in the set of whitespace-preserving names is determined by the best match among all the `xsl:strip-space` or `xsl:preserve-space` declarations: it is included if and only if there is no match or the best match is an `xsl:preserve-space` element. The `xsl:strip-space` and `xsl:preserve-space` elements each have an `elements` attribute whose value is a whitespace-separated list of NameTests [XP]; an element name matches an `xsl:strip-space` or `xsl:preserve-space` element if it matches one of the NameTests [XP]. An element matches a NameTest [XP] if and only if the NameTest [XP] would be true for the element as an XPath node test. When more than one `xsl:strip-space` and `xsl:preserve-space` element matches, the best matching element is determined by the best matching NameTest [XP]. This is determined in the same way as with template rules:

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

[ERR XTRE0270] It is a recoverable dynamic error if this leaves more than one match, unless all the matched declarations are equivalent (that is, they are all `xsl:strip-space` or they are all `xsl:preserve-space`). The optional recovery action is to select, from the matches that are left, the one that occurs last in declaration order.

If an element in a source document has a type annotation that is a simple type or a complex type with simple content, then any whitespace text nodes among its children are preserved, regardless of any `xsl:strip-space` declarations. The reason for this is that stripping a whitespace text node from an element with simple content could make the element invalid: for example, it could cause the `minLength` facet to be violated.

Stripping of type annotations happens before stripping of whitespace text nodes, so this situation will not occur if `input-type-annotations="strip"` is specified.

Note:

In [Data Model], processes are described for constructing an XDM tree from an Infoset or from a PSVI. Those processes deal with whitespace according to their own rules, and the provisions in this section apply to the resulting tree. In practice this means that elements that are defined in a DTD or a Schema to contain element-only content will have whitespace text nodes stripped, regardless of the `xsl:strip-space` and `xsl:preserve-space` declarations in the stylesheet.

However, source trees are not necessarily constructed using those processes; indeed, they are not necessarily constructed by parsing XML documents. Nothing in the XSLT specification constrains how the source tree is constructed, or what happens to whitespace text nodes during its construction. The provisions in this section relate only to whitespace text nodes that are present in the tree supplied as input to the XSLT processor. The XSLT processor cannot preserve whitespace text nodes unless they were actually present in the supplied tree.

4.5 Attribute Types and DTD Validation

The mapping from the Infoset to the XDM data model, described in [Data Model], does not retain attribute types. This means, for example, that an attribute described in the DTD as having attribute type `NMTOKENS` will be annotated in the XDM tree as `xs:untypedAtomic` rather than `xs:NMTOKENS`, and its typed value will consist of a single `xs:untypedAtomic` value rather than a sequence of `xs:NMTOKEN` values.

Attributes with a DTD-derived type of ID, IDREF, or IDREFS will be marked in the XDM tree as having the `is-id` or `is-idrefs` properties. It is these properties, rather than any type annotation, that are examined by the functions `id` [FO] and `idref` [FO] described in [Functions and Operators].

4.6 Limits

The XDM data model (see [Data Model]) leaves it to the host language to define limits. This section describes the limits that apply to XSLT.

Limits on some primitive data types are defined in [XML Schema Part 2]. Other limits, listed below, are implementation-defined. Note that this does not necessarily mean that each limit must be a simple constant: it may vary depending on environmental factors such as available resources.

The following limits are implementation-defined:

1. For the `xs:decimal` type, the maximum number of decimal digits (the `totalDigits` facet). This must be at least 18 digits. (Note, however, that support for the full value range of `xs:unsignedLong` requires 20 digits.)
2. For the types `xs:date`, `xs:time`, `xs:dateTime`, `xs:gYear`, and `xs:gYearMonth`: the range of values of the year component, which must be at least +0001 to +9999; and the maximum number of fractional second digits, which must be at least 3.
3. For the `xs:duration` type: the maximum absolute values of the years, months, days, hours, minutes, and seconds components.
4. For the `xs:yearMonthDuration` type: the maximum absolute value, expressed as an integer number of months.
5. For the `xs:dayTimeDuration` type: the maximum absolute value, expressed as a decimal number of seconds.
6. For the types `xs:string`, `xs:hexBinary`, `xs:base64Binary`, `xs:QName`, `xs:anyURI`, `xs:NOTATION`, and types derived from them: the maximum length of the value.
7. For sequences, the maximum number of items in a sequence.

4.7 Disable Output Escaping

For backwards compatibility reasons, XSLT 2.0 continues to support the `disable-output-escaping` feature introduced in XSLT 1.0. This is an optional feature and implementations are not REQUIRED to support it. A new facility, that of named character maps (see *20.1 Character*

Maps) is introduced in XSLT 2.0. It provides similar capabilities to `disable-output-escaping`, but without distorting the data model.

If an implementation supports the `disable-output-escaping` attribute of `xsl:text` and `xsl:value-of`, (see *20.2 Disabling Output Escaping*), then the data model for trees constructed by the processor is augmented with a boolean value representing the value of this property. This boolean value, however, can be set only within a final result tree that is being passed to the serializer.

Conceptually, each character in a text node on such a result tree has a boolean property indicating whether the serializer is to disable the normal rules for escaping of special characters (for example, outputting of `&` as `&`) in respect of this character or attribute node.

Note:

In practice, the nodes in a final result tree will often be streamed directly from the XSLT processor to the serializer. In such an implementation, `disable-output-escaping` can be viewed not so much a property stored with nodes in the tree, but rather as additional information passed across the interface between the XSLT processor and the serializer.

5 Features of the XSLT Language

5.1 Qualified Names

The name of a stylesheet-defined object, specifically a named template, a mode, an attribute set, a key, a decimal-format, a variable or parameter, a stylesheet function, a named output definition, or a character map is specified as a QName using the syntax for QName [Names] as defined in [Namespaces in XML 1.0].

[DEFINITION: A **QName** is always written in the form `(NCName ":")? NCName`, that is, a local name optionally preceded by a namespace prefix. When two QNames are compared, however, they are considered equal if the corresponding expanded-QNames are the same, as described below.]

Because an atomic value of type `xs:QName` is sometimes referred to loosely as a QName, this specification also uses the term lexical QName to emphasize that it is referring to a QName [Names] in its lexical form rather than its expanded form. This term is used especially when strings containing lexical QNames are manipulated as run-time values.

[DEFINITION: A **lexical QName** is a string representing a QName in the form `(NCName ":")? NCName`, that is, a local name optionally preceded by a namespace prefix.]

[DEFINITION: A string in the form of a lexical QName may occur as the value of an attribute node in a stylesheet module, or within an XPath expression contained in such an attribute node, or as the result of evaluating an XPath expression contained in such an attribute node. The element containing this attribute node is referred to as the **defining element** of the QName.]

[DEFINITION: An **expanded-QName** contains a pair of values, namely a local name and an optional namespace URI. It may also contain a namespace prefix. Two expanded-QNames are equal if the namespace URIs are the same (or both absent) and the local names are the same. The prefix plays no part in the comparison, but is used only if the expanded-QName needs to be converted back to a string.]

If the QName has a prefix, then the prefix is expanded into a URI reference using the namespace declarations in effect on its defining element. The expanded-QName 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 of the defining element (see Section 6.2 Element Nodes[DM]) is *not* used for unprefixed names.

There are three cases where the default namespace of the defining element *is* used when expanding an unprefixed QName:

1. Where a QName is used to define the name of an element being constructed. This applies both to cases where the name is known statically (that is, the name of a literal result element) and to cases where it is computed dynamically (the value of the `name` attribute of the `xsl:element` instruction).
2. The default namespace is used when expanding the first argument of the function `element-available`.
3. The default namespace applies to any unqualified element names appearing in the `cdata-section-elements` attribute of `xsl:output` or `xsl:result-document`

In the case of an unprefixed QName used as a `NameTest` within an XPath expression (see *5.3 Expressions*) , and in certain other contexts, the namespace to be used in expanding the QName may be specified by means of the `[xsl:]xpath-default-namespace` attribute, as specified in *5.2 Unprefixed QNames in Expressions and Patterns*.

[ERR XTSE0280] In the case of a prefixed QName used as the value of an attribute in the stylesheet, or appearing within an XPath expression in the stylesheet, it is a static error if the defining element has no namespace node whose name matches the prefix of the QName.

[ERR XTDE0290] Where the result of evaluating an XPath expression (or an attribute value template) is required to be a lexical QName, then unless otherwise specified it is a non-recoverable dynamic error if the defining element has no namespace node whose name matches the prefix of the lexical QName. This error MAY be signaled as a static error if the value of the expression can be determined statically.

5.2 Unprefixed QNames in Expressions and Patterns

The attribute `[xsl:]xpath-default-namespace` (see *3.5 Standard Attributes*) may be used on an element in the stylesheet to define the namespace that will be used for an unprefixed element name or type name within an XPath expression, and in certain other contexts listed below.

The value of the attribute is the namespace URI to be used.

For any element in the stylesheet, this attribute has an effective value, which is the value of the `[xsl:]xpath-default-namespace` on that element or on the innermost containing element that specifies such an attribute, or the zero-length string if no containing element specifies such an attribute.

For any element in the stylesheet, the effective value of this attribute determines the value of the *default namespace for element and type names* in the static context of any XPath expression contained in an attribute of that element (including XPath expressions in attribute value templates). The effect of this is specified in [XPath 2.0]; in summary, it determines the namespace used for any unprefixed type name in the SequenceType production, and for any element name appearing in a path expression or in the SequenceType production.

The effective value of this attribute similarly applies to any of the following constructs appearing within its scope:

- any unprefixed element name or type name used in a pattern
- any unprefixed element name used in the `elements` attribute of the `xsl:strip-space` or `xsl:preserve-space` instructions
- any unprefixed element name or type name used in the `as` attribute of an XSLT element
- any unprefixed type name used in the `type` attribute of an XSLT element
- any unprefixed type name used in the `xsl:type` attribute of a literal result element.

The `[xsl:]xpath-default-namespace` attribute MUST be in the XSLT namespace if and only if its parent element is *not* in the XSLT namespace.

If the effective value of the attribute is a zero-length string, which will be the case if it is explicitly set to a zero-length string or if it is not specified at all, then an unprefixed element name or type name refers to a name that is in no namespace. The default namespace of the parent element (see Section 6.2 Element Nodes[DM]) is *not* used.

The attribute does not affect other names, for example function names, variable names, or template names, or strings that are interpreted as lexical QNames during stylesheet evaluation, such as the effective value of the `name` attribute of `xsl:element` or the string supplied as the first argument to the `key` function.

5.3 Expressions

XSLT uses the expression language defined by XPath 2.0 [XPath 2.0]. 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 a result tree.

[DEFINITION: Within this specification, the term **XPath expression**, or simply **expression**, means a string that matches the production Expr [XP] defined in [XPath 2.0].]

An XPath expression may occur as the value of certain attributes on XSLT-defined elements, and also within curly brackets in attribute value templates.

Except where forwards-compatible behavior is enabled (see *3.9 Forwards-Compatible Processing*), it is a static error if the value of such an attribute, or the text between curly brackets in an attribute value template, does not match the XPath production Expr [XP], or if it fails to satisfy other static constraints defined in the XPath specification, for example that all variable references MUST refer to variables that are in scope. Error codes are defined in [XPath 2.0].

The transformation fails with a non-recoverable dynamic error if any XPath expression is evaluated and raises a dynamic error. Error codes are defined in [XPath 2.0].

The transformation fails with a type error if an XPath expression raises a type error, or if the result of evaluating the XPath expression is evaluated and raises a type error, or if the XPath processor signals a type error during static analysis of an expression. Error codes are defined in [XPath 2.0].

[DEFINITION: The context within a stylesheet where an XPath expression appears may specify the **required type** of the expression. The required type indicates the type of the value that the expression is expected to return.] If no required type is specified, the expression may return any value: in effect, the required type is then `item()*`.

[DEFINITION: Except where otherwise indicated, the actual value of an expression is converted to the required type using the **function conversion rules**. These are the rules defined in [XPath 2.0] for converting the supplied argument of a function call to the required type of that argument, as defined in the function signature. The relevant rules are those that apply when XPath 1.0 compatibility mode is set to `false`.]

This specification also invokes the XPath 2.0 function conversion rules to convert the result of evaluating an XSLT sequence constructor to a required type (for example, the sequence constructor enclosed in an `xsl:variable`, `xsl:template`, or `xsl:function` element).

Any dynamic error or type error that occurs when applying the function conversion rules to convert a value to a required type results in the transformation failing, in the same way as if the error had occurred while evaluating an expression.

> **Note:**
>
> Note the distinction between the two kinds of error that may occur. Attempting to convert an integer to a date is a type error, because such a conversion is never possible. Type errors can be reported statically if they can be detected statically, whether or not the construct in question is ever evaluated. Attempting to convert the string `2003-02-29` to a date is a dynamic error rather than a type error, because the problem is with this particular value, not with its type. Dynamic errors are reported only if the instructions or expressions that cause them are actually evaluated.

5.4 The Static and Dynamic Context

XPath defines the concept of an expression context[XP] which contains all the information that can affect the result of evaluating an expression. The expression context has two parts, the static context[XP], and the dynamic context[XP]. The components that make up the expression context are defined in the XPath specification (see Section 2.1 Expression Context[XP]). This section describes the way in which these components are initialized when an XPath expression is contained within an XSLT stylesheet.

As well as providing values for the static and dynamic context components defined in the XPath specification, XSLT defines additional context components of its own. These context components are used by XSLT instructions (for example, `xsl:next-match` and `xsl:apply-imports`), and also by the functions in the extended function library described in this specification.

The following four sections describe:

5.4.1 Initializing the Static Context

5.4.1 Initializing the Static Context

The static context[XP] of an XPath expression appearing in an XSLT stylesheet is initialized as follows. In these rules, the term **containing element** means the element within the stylesheet that is the parent of the attribute whose value contains the XPath expression in question, and the term **enclosing element** means the containing element or any of its ancestors.

- XPath 1.0 compatibility mode is set to true if and only if the containing element occurs in part of the stylesheet where backwards compatible behavior is enabled (see *3.8 Backwards-Compatible Processing*).
- The statically known namespaces[XP] are the namespace declarations that are in scope for the containing element.
- The default element/type namespace[XP] is the namespace defined by the `[xsl:]xpath-default-namespace` attribute on the innermost enclosing element that has such an attribute, as described in *5.2 Unprefixed QNames in Expressions and Patterns*. The value of this attribute is a namespace URI. If there is no `[xsl:]xpath-default-namespace` attribute on an enclosing element, the default namespace for element names and type names is the null namespace.
- The default function namespace[XP] is the standard function namespace, defined in [Functions and Operators]. This means that it is not necessary to declare this namespace in the stylesheet, nor is it necessary to use the prefix `fn` (or any other prefix) in calls to the core functions.
- The in-scope schema definitions[XP] for the XPath expression are the same as the in-scope schema components for the stylesheet, and are as specified in *3.13 Built-in Types*.
- The in-scope variables[XP] are defined by the variable binding elements that are in scope for the containing element (see *9 Variables and Parameters*).
- The function signatures[XP] are the core functions defined in [Functions and Operators], the constructor functions for all the atomic types in the in-scope schema definitions[XP], the additional functions defined in this specification, the stylesheet functions defined in the stylesheet, plus any extension functions bound using implementation-defined mechanisms (see *18 Extensibility and Fallback*).

  Note:

  It follows from the above that a conformant XSLT processor must implement the entire library of core functions defined in [Functions and Operators].
- The statically known collations[XP] are implementation-defined. However, the set of in-scope collations MUST always include the Unicode codepoint collation, defined in Section 7.3 Equality and Comparison of Strings[FO].
- The default collation[XP] is defined by the value of the `[xsl:]default-collation` attribute on the innermost enclosing element that has such an attribute. For details, see *3.6.1 The default-collation attribute*.

  [DEFINITION: In this specification the term **default collation** means the collation that is used by XPath operators such as `eq` and `lt` appearing in XPath expressions within the stylesheet.]

  This collation is also used by default when comparing strings in the evaluation of the `xsl:key` and `xsl:for-each-group` elements. This MAY also (but need not necessarily) be the same as the default collation used for `xsl:sort` elements within the stylesheet. Collations used by `xsl:sort` are described in *13.1.3 Sorting Using Collations*.
- The base URI[XP] is the base URI of the containing element. The concept of the base URI of a node is defined in Section 5.2 base-uri Accessor[DM]

5.4.2 Additional Static Context Components used by XSLT

Some of the components of the XPath static context are used also by XSLT elements. For example, the `xsl:sort` element makes use of the collations defined in the static context, and attributes such as `type` and `as` may reference types defined in the in-scope schema components.

Many top-level declarations in a stylesheet, and attributes on the `xsl:stylesheet` element, affect the behavior of instructions within the stylesheet. Each of these constructs is described in its appropriate place in this specification.

A number of these constructs are of particular significance because they are used by functions defined in XSLT, which are added to the library of functions available for use in XPath expressions within the stylesheet. These are:

- The set of named keys, used by the `key` function
- The set of named decimal formats, used by the `format-number` function
- The values of system properties, used by the `system-property` function
- The set of available instructions, used by the `element-available` function

5.4.3 Initializing the Dynamic Context

For convenience, the dynamic context is described in two parts: the focus, which represents the place in the source document that is currently being processed, and a collection of additional context variables.

A number of functions specified in [Functions and Operators] are defined to be stable [FO], meaning that if they are called twice during the same execution scope[FO], with the same arguments, then they return the same results (see Section 1.7 Terminology[FO]). In XSLT, the execution of a stylesheet defines the execution scope. This means, for example, that if the function `current-dateTime`[FO] is called repeatedly during a transformation, it produces the same result each time. By implication, the components of the dynamic context on which these functions depend are also stable for the duration of the transformation. Specifically, the following components defined in Section 2.1.2 Dynamic Context[XP] must be stable: *function implementations*, *current dateTime*, *implicit timezone*, *available documents*, *available collections*, and *default collection*. The values of global variables and stylesheet parameters are also stable for the duration of a transformation. The focus is *not* stable; the additional dynamic context components defined in *5.4.4 Additional Dynamic Context Components used by XSLT* are also *not* stable.

As specified in [Functions and Operators], implementations may provide user options that relax the requirement for the `doc` [FO] and `collection`[FO] functions (and therefore, by implication, the `document` function) to return stable results. By default, however, the functions must be stable. The manner in which such user options are provided, if at all, is implementation-defined.

XPath expressions contained in `[xsl:]use-when` attributes are not considered to be evaluated "during the transformation" as defined above. For details see *3.12 Conditional Element Inclusion*.

5.4.3.1 Maintaining Position: the Focus

[DEFINITION: When a sequence constructor is evaluated, the processor keeps track of which items are being processed by means of a set of implicit variables referred to collectively as the **focus**.] More specifically, the focus consists of the following three values:

- [DEFINITION: The **context item** is the item currently being processed. An item (see [Data Model]) is either an atomic value (such as an integer, date, or string), or a node. The context item is initially set to the initial context node supplied when the transformation is invoked (see *2.3 Initiating a Transformation*). It changes whenever instructions such as `xsl:apply-templates` and `xsl:for-each` are used to process a sequence of items; each item in such a sequence becomes the context item while that item is being processed.] The context item is returned by the XPath expression `.` (dot).
- [DEFINITION: The **context position** is the position of the context item within the sequence of items currently being processed. It changes whenever the context item changes. When an instruction such as `xsl:apply-templates` or `xsl:for-each` is used to process a sequence of items, the first item in the sequence is processed with a context position of 1, the second item with a context position of 2, and so on.] The context position is returned by the XPath expression `position()`.
- [DEFINITION: The **context size** is the number of items in the sequence of items currently being processed. It changes whenever instructions such as `xsl:apply-templates` and `xsl:for-each` are used to process a sequence of items; during the processing of each one of those items, the context size is set to the count of the number of items in the sequence (or equivalently, the position of the last item in the sequence).] The context size is returned by the XPath expression `last()`.

[DEFINITION: If the context item is a node (as distinct from an atomic value such as an integer), then it is also referred to as the **context node**. The context node is not an independent variable, it changes whenever the context item changes. When the context item is an atomic value, there is no context node.] The context node is returned by the XPath expression `self::node()`, and it is used as the starting node for all relative path expressions.

Where the containing element of an XPath expression is an instruction or a literal result element, the initial context item, context position, and context size for the XPath expression are the same as the context item, context position, and context size for the evaluation of the containing instruction or literal result element.

In other cases (for example, where the containing element is `xsl:sort`, `xsl:with-param`, or `xsl:key`), the rules are given in the specification of the containing element.

The `current` function can be used within any XPath expression to select the item that was supplied as the context item to the XPath expression by the XSLT processor. Unlike `.` (dot) this is unaffected by changes to the context item that occur within the XPath expression. The `current` function is described in *16.6.1 current*.

On completion of an instruction that changes the focus (such as `xsl:apply-templates` or `xsl:for-each`), the focus reverts to its previous value.

When a stylesheet function is called, the focus within the body of the function is initially undefined. The focus is also undefined on initial entry to the stylesheet if no initial context node is supplied.

When the focus is undefined, evaluation of any expression that references the context item, context position, or context size results in a non-recoverable dynamic error [XPDY0002]

The description above gives an outline of the way the focus works. Detailed rules for the effect of each instruction are given separately with the description of that instruction. In the absence of specific rules, an instruction uses the same focus as its parent instruction.

[DEFINITION: A **singleton focus** based on a node *N* has the context item (and therefore the context node) set to *N*, and the context position and context size both set to 1 (one).]

5.4.3.2 Other components of the XPath Dynamic Context

The previous section explained how the focus for an XPath expression appearing in an XSLT stylesheet is initialized. This section explains how the other components of the dynamic context[XP] of an XPath expression are initialized.

- The dynamic variables[XP] are the current values of the in-scope variable binding elements.
- The *current date and time* represents an implementation-dependent point in time during processing of the transformation; it does not change during the course of the transformation.
- The implicit timezone[XP] is implementation-defined.
- The available documents[XP], and the available collections[XP] are determined as part of the process for initiating a transformation (see *2.3 Initiating a Transformation*).

  The available documents[XP] are defined as part of the XPath 2.0 dynamic context to support the `doc`[FO] function, but this component is also referenced by the similar XSLT `document` function: see *16.1 Multiple Source Documents*. This variable defines a mapping between URIs passed to the `doc`[FO] or `document` function and the document nodes that are returned.

  Note:

  Defining this as part of the evaluation context is a formal way of specifying that the way in which URIs get turned into document nodes is outside the control of the language specification, and depends entirely on the run-time environment in which the transformation takes place.

  The XSLT-defined `document` function allows the use of URI references containing fragment identifiers. The interpretation of a fragment identifier depends on the media type of the resource representation. Therefore, the information supplied in available documents[XP] for XSLT processing must provide not only a mapping from URIs to document nodes as required by XPath, but also a mapping from URIs to media types.
- The default collection[XP] is implementation-defined. This allows options such as setting the default collection to be an empty sequence, or to be undefined.

5.4.4 Additional Dynamic Context Components used by XSLT

In addition to the values that make up the focus, an XSLT processor maintains a number of other dynamic context components that reflect

aspects of the evaluation context. These components are fully described in the sections of the specification that maintain and use them. They are:

- The current template rule, which is the template rule most recently invoked by an `xsl:apply-templates`, `xsl:apply-imports`, or `xsl:next-match` instruction: see *6.7 Overriding Template Rules*;
- The current mode, which is the mode set by the most recent call of `xsl:apply-templates` (for a full definition see *6.5 Modes*);
- The current group and current grouping key, which provide information about the collection of items currently being processed by an `xsl:for-each-group` instruction: see *14.1 The Current Group* and *14.2 The Current Grouping Key*;
- The current captured substrings: this is a sequence of strings, which is maintained when a string is matched against a regular expression using the `xsl:analyze-string` instruction, and which is accessible using the `regex-group` function: see *15.2 Captured Substrings*.
- The output state: this is a flag whose two possible values are final output state and temporary output state. This flag indicates whether instructions are currently writing to a final result tree or to an internal data structure. The initial setting is final output state, and it is switched to temporary output state by instructions such as `xsl:variable`. For more details, see *19.1 Creating Final Result Trees*.

The following non-normative table summarizes the initial state of each of the components in the evaluation context, and the instructions which cause the state of the component to change.

Component	Initial Setting	Set by	Cleared by
focus	singleton focus based on the initial context node if supplied	`xsl:apply-templates`, `xsl:for-each`, `xsl:for-each-group`, `xsl:analyze-string`	calls on stylesheet functions
current template rule	If a named template is supplied as the entry point to the transformation, then null; otherwise the initial template	`xsl:apply-templates`, `xsl:apply-imports`, `xsl:next-match`	`xsl:for-each`, `xsl:for-each-group`, and `xsl:analyze-string`, and calls on stylesheet functions. Also cleared while evaluating global variables or default values of stylesheet parameters, and the sequence constructors contained in `xsl:key` and `xsl:sort`.
current mode	the initial mode	`xsl:apply-templates`	calls on stylesheet functions
current group	empty sequence	`xsl:for-each-group`	calls on stylesheet functions
current grouping key	empty sequence	`xsl:for-each-group`	calls on stylesheet functions
current captured substrings	empty sequence	`xsl:matching-substring`	`xsl:non-matching-substring`; calls on stylesheet functions
output state	final output state	Set to temporary output state by instructions such as `xsl:variable`, `xsl:attribute`, etc., and by calls on stylesheet functions	None

5.5 Patterns

A template rule identifies the nodes to which it applies by means of a pattern. As well as being used in template rules, patterns are used for numbering (see *12 Numbering*), for grouping (see *14 Grouping*), and for declaring keys (see *16.3 Keys*).

[DEFINITION: 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.] As explained in detail below, a node matches a pattern if the node can be selected by deriving an equivalent expression, and evaluating this expression with respect to some possible context.

5.5.1 Examples of Patterns

Example: Patterns

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/entry` matches any `entry` element with an `olist` parent.
- `appendix//para` matches any `para` element with an `appendix` ancestor element.
- `schema-element(us:address)` matches any element that is annotated as an instance of the type defined by the schema element declaration `us:address`, and whose name is either `us:address` or the name of another element in its substitution group.
- `attribute(*, xs:date)` matches any attribute annotated as being of type `xs:date`.
- `/` matches a document node.
- `document-node()` matches a document node.
- `document-node(schema-element(my:invoice))` matches the document node of a document whose document element is named `my:invoice` and matches the type defined by the global element declaration `my:invoice`.
- `text()` matches any text node.
- `node()` matches any node other than an attribute node, namespace node, or document node.
- `id("W33")` matches the element with unique ID `W33`.
- `para[1]` matches any `para` element that is the first `para` child element of its parent. It also matches a parentless `para` element.

- `//para` matches any `para` element that has a parent node.
- `bullet[position() mod 2 = 0]` matches any `bullet` element that is an even-numbered `bullet` 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 node.

5.5.2 Syntax of Patterns

[ERR XTSE0340] Where an attribute is defined to contain a pattern, it is a static error if the pattern does not match the production Pattern. Every pattern is a legal XPath expression, but the converse is not true: `2+2` is an example of a legal XPath expression that is not a pattern. The XPath expressions that can be used as patterns are those that match the grammar for Pattern, given below.

Informally, a Pattern is a set of path expressions separated by `|`, where each step in the path expression is constrained to be an AxisStep [XP] that uses only the `child` or `attribute` axes. Patterns may also use the `//` operator. A Predicate [XP] within the PredicateList [XP] in a pattern can contain arbitrary XPath expressions (enclosed between square brackets) in the same way as a predicate [XP] in a path expression.

Patterns may start with an `id` [FO] or `key` function call, provided that the value to be matched is supplied as either a literal or a reference to a variable or parameter, and the key name (in the case of the `key` function) is supplied as a string literal. These patterns will never match a node in a tree whose root is not a document node.

If a pattern occurs in part of the stylesheet where backwards compatible behavior is enabled (see *3.8 Backwards-Compatible Processing*), then the semantics of the pattern are defined on the basis that the equivalent XPath expression is evaluated with XPath 1.0 compatibility mode set to true.

Patterns

```
[1]  Pattern             ::=  PathPattern
                              | Pattern '|' PathPattern
[2]  PathPattern         ::=  RelativePathPattern
                              | '/' RelativePathPattern?
                              | '//' RelativePathPattern
                              | IdKeyPattern (('/' | '//') RelativePathPattern)?
[3]  RelativePathPattern ::=  PatternStep (('/' | '//') RelativePathPattern)?
[4]  PatternStep         ::=  PatternAxis? NodeTest [XP] PredicateList[XP]
[5]  PatternAxis         ::=  ('child' '::' | 'attribute' '::' | '@')
[6]  IdKeyPattern        ::=  'id' '(' IdValue ')'
                              | 'key' '(' StringLiteral[XP] ',' KeyValue ')'
[7]  IdValue             ::=  StringLiteral[XP] | VarRef [XP]
[8]  KeyValue            ::=  Literal [XP] | VarRef [XP]
```

The constructs NodeTest [XP], PredicateList [XP], VarRef [XP], Literal [XP], and StringLiteral [XP] are part of the XPath expression language, and are defined in [XPath 2.0].

5.5.3 The Meaning of a Pattern

The meaning of a pattern is defined formally as follows.

First we define the concept of an *equivalent expression*. In general, the equivalent expression is the XPath expression that takes the same lexical form as the pattern as written. However, if the pattern contains a `PathPattern` that is a `RelativePathPattern`, then the first `PatternStep` *PS* of this `RelativePathPattern` is adjusted to allow it to match a parentless element or attribute node, as follows:

- If the `NodeTest` in *PS* is `document-node()` (optionally with arguments), and if no explicit axis is specified, then the axis in step *PS* is taken as `self` rather than `child`.
- If *PS* uses the child axis (explicitly or implicitly), and if the `NodeTest` in *PS* is not `document-node()` (optionally with arguments), then the axis in step *PS* is replaced by `child-or-top`, which is defined as follows. If the context node is a parentless element, comment, processing-instruction, or text node then the `child-or-top` axis selects the context node; otherwise it selects the children of the context node. It is a forwards axis whose principal node kind is element.
- If *PS* uses the attribute axis, then the axis in step *PS* is replaced by `attribute-or-top`, which is defined as follows. If the context node is an attribute node with no parent, then the `attribute-or-top` axis selects the context node; otherwise it selects the attributes of the context node. It is a forwards axis whose principal node kind is attribute.

The axes `child-or-top` and `attribute-or-top` are introduced only for definitional purposes. They cannot be used explicitly in a user-written pattern or expression.

> **Note:**
>
> The purpose of these adjustments is to ensure that a pattern such as `person` matches any element named `person`, even if it has no parent; and similarly, that the pattern `@width` matches any attribute named `width`, even a parentless attribute. The rule also ensures that a pattern using a `NodeTest` of the form `document-node(...)` matches a document node. The pattern `node()` will match any element, text node, comment, or processing instruction, whether or not it has a parent. For backwards compatibility reasons, the pattern `node()`, when used without an explicit axis, does not match document nodes, attribute nodes, or namespace nodes. The rules are also phrased to ensure that positional patterns of the form `para[1]` continue to count nodes relative to their parent, if they have one.

Let the equivalent expression, calculated according to these rules, be *EE*.

To determine whether a node *N* matches the pattern, evaluate the expression `root(.)//(EE)` with a singleton focus based on *N*. If the result is a sequence of nodes that includes *N*, then node *N* matches the pattern; otherwise node *N* does not match the pattern.

Example: The Semantics of Patterns

The pattern `p` matches any `p` element, because a `p` element will always be present in the result of evaluating the expression `root(.)//(child-or-top::p)`. Similarly, `/` matches a document node, and only a document node, because the result of the expression `root(.)//(/)` returns the root node of the tree containing the context node if and only if it is a document node.

The pattern `node()` matches all nodes selected by the expression `root(.)//(child-or-top::node())`, that is, all element, text, comment, and processing instruction nodes, whether or not they have a parent. It does not match attribute or namespace nodes because the expression does not select nodes using the attribute or namespace axes. It does not match document nodes because for backwards compatibility reasons the `child-or-top` axis does not match a document node.

Although the semantics of patterns are specified formally in terms of expression evaluation, it is possible to understand pattern matching using a different model. In a pattern, `|` indicates alternatives; a pattern with one or more `|` separated alternatives matches if any one of the alternatives matches. A pattern such as `book/chapter/section` can be examined from right to left. A node will only match this pattern if it is a `section` element; and then, only if its parent is a `chapter`; and then, only if the parent of that `chapter` is a `book`. When the pattern uses the `//` operator, one can still read it from right to left, but this time testing the ancestors of a node rather than its parent. For example `appendix//section` matches every `section` element that has an ancestor `appendix` element.

The formal definition, however, is useful for understanding the meaning of a pattern such as `para[1]`. This matches any node selected by the expression `root(.)//(child-or-top::para[1])`: that is, any `para` element that is the first `para` child of its parent, or a `para` element that has no parent.

Note:

An implementation, of course, may use any algorithm it wishes for evaluating patterns, so long as the result corresponds with the formal definition above. An implementation that followed the formal definition by evaluating the equivalent expression and then testing the membership of a specific node in the result would probably be very inefficient.

5.5.4 Errors in Patterns

Any dynamic error or type error that occurs during the evaluation of a pattern against a particular node is treated as a recoverable error even if the error would not be recoverable under other circumstances. The optional recovery action is to treat the pattern as not matching that node.

Note:

The reason for this provision is that it is difficult for the stylesheet author to predict which predicates in a pattern will actually be evaluated. In the case of match patterns in template rules, it is not even possible to predict which patterns will be evaluated against a particular node. Making errors in patterns recoverable enables an implementation, if it chooses to do so, to report such errors while stylesheets are under development, while masking them if they occur during production running.

One particular optimization is REQUIRED by this specification: for a PathPattern that starts with `/` or `//` or with an IdKeyPattern, the result of testing this pattern against a node in a tree whose root is not a document node must be a non-match, rather than a dynamic error. This rule applies to each PathPattern within a Pattern.

Note:

Without the above rule, any attempt to apply templates to a parentless element node would create the risk of a dynamic error if the stylesheet has a template rule specifying `match="/"`.

5.6 Attribute Value Templates

[DEFINITION: In an attribute that is designated 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 brackets (`{}`)].

An attribute value template consists of an alternating sequence of fixed parts and variable parts. A variable part consists of an XPath expression enclosed in curly brackets (`{}`). A fixed part may contain any characters, except that a left curly bracket MUST be written as `{{` and a right curly bracket MUST be written as `}}`.

Note:

An expression within a variable part may contain an unescaped curly bracket within a StringLiteral [XP] or within a comment.

[ERR XTSE0350] It is a static error if an unescaped left curly bracket appears in a fixed part of an attribute value template without a matching right curly bracket.

It is a static error if the string contained between matching curly brackets in an attribute value template does not match the XPath production Expr [XP], or if it contains other XPath static errors. The error is signaled using the appropriate XPath error code.

[ERR XTSE0370] It is a static error if an unescaped right curly bracket occurs in a fixed part of an attribute value template.

[DEFINITION: The result of evaluating an attribute value template is referred to as the **effective value** of the attribute.] The effective value is the string obtained by concatenating the expansions of the fixed and variable parts:

- The expansion of a fixed part is obtained by replacing any double curly brackets (`{{` or `}}`) by the corresponding single curly bracket.
- The expansion of a variable part is obtained by evaluating the enclosed XPath expression and converting the resulting value to a string. This conversion is done using the rules given in *5.7.2 Constructing Simple Content*.

Note:

This process can generate dynamic errors, for example if the sequence contains an element with a complex content type (which cannot

be atomized).

If backwards compatible behavior is enabled for the attribute, the rules for converting the value of the expression to a string are modified as follows. After atomizing the result of the expression, all items other than the first item in the resulting sequence are discarded, and the effective value is obtained by converting the first item in the sequence to a string. If the atomized sequence is empty, the result is a zero-length string.

Curly brackets are not treated specially in an attribute value in an XSLT stylesheet unless the attribute is specifically designated as one that permits an attribute value template; in an element syntax summary, the value of such attributes is surrounded by curly brackets.

Note:

Not all attributes are designated as attribute value templates. Attributes whose value is an expression or pattern, attributes of declaration elements and attributes that refer to named XSLT objects are generally not designated as attribute value templates (an exception is the `format` attribute of `xsl:result-document`). Namespace declarations are not XDM attribute nodes and are therefore never treated as attribute value templates.

Example: Attribute Value Templates

The following example creates an `img` result element from a `photograph` element in the source; the value of the `src` and `width` attributes are computed using XPath expressions enclosed in attribute value templates:

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

<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"/>
```

Example: Producing a Space-Separated List

The following example shows how the values in a sequence are output as a space-separated list. The following literal result element:

```
<temperature readings="{10.32, 5.50, 8.31}"/>
```

produces the output node:

```
<temperature readings="10.32 5.5 8.31"/>
```

Curly brackets are *not* recognized recursively inside expressions.

Example: Curly Brackets can not be Nested

For example:

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

is *not* allowed. Instead, use simply:

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

5.7 Sequence Constructors

[DEFINITION: A **sequence constructor** is a sequence of zero or more sibling nodes in the stylesheet that can be evaluated to return a sequence of nodes and atomic values. The way that the resulting sequence is used depends on the containing instruction.]

Many XSLT elements, and also literal result elements, are defined to take a sequence constructor as their content.

Four kinds of nodes may be encountered in a sequence constructor:

- *Text nodes* appearing in the stylesheet (if they have not been removed in the process of whitespace stripping: see *4.2 Stripping Whitespace from the Stylesheet*) are copied to create a new parentless text node in the result sequence.

- Literal result elements are evaluated to create a new parentless element node, having the same expanded-QName as the literal result element, which is added to the result sequence: see *11.1 Literal Result Elements*
- XSLT instructions produce a sequence of zero, one, or more items as their result. These items are added to the result sequence. For most XSLT instructions, these items are nodes, but some instructions (`xsl:sequence` and `xsl:copy-of`) can also produce atomic values. Several instructions, such as `xsl:element`, return a newly constructed parentless node (which may have its own attributes, namespaces, children, and other descendants). Other instructions, such as `xsl:if`, pass on the items produced by their own nested sequence constructors. The `xsl:sequence` instruction may return atomic values, or existing nodes.
- Extension instructions (see *18.2 Extension Instructions*) also produce a sequence of items as their result. The items in this sequence are added to the result sequence.

There are several ways the result of a sequence constructor may be used.

- The sequence may be bound to a variable or returned from a stylesheet function, in which case it becomes available as a value to be manipulated in arbitrary ways by XPath expressions. The sequence is bound to a variable when the sequence constructor appears within one of the elements `xsl:variable`, `xsl:param`, or `xsl:with-param`, when this instruction has an `as` attribute. The sequence is returned from a stylesheet function when the sequence constructor appears within the `xsl:function` element.

  Note:

  This will typically expose to the stylesheet elements, attributes, and other nodes that have not yet been attached to a parent node in a result tree. The semantics of XPath expressions when applied to parentless nodes are well-defined; however, such expressions should be used with care. For example, the expression `/` causes a type error if the root of the tree containing the context node is not a document node..

  Parentless attribute nodes require particular care because they have no namespace nodes associated with them. A parentless attribute node is not permitted to contain namespace-sensitive content (for example, a QName or an XPath expression) because there is no information enabling the prefix to be resolved to a namespace URI. Parentless attributes can be useful in an application (for example, they provide an alternative to the use of attribute sets: see *10.2 Named Attribute Sets*) but they need to be handled with care.
- The sequence may be returned as the result of the containing element. This happens when the instruction containing the sequence constructor is `xsl:analyze-string`, `xsl:apply-imports`, `xsl:apply-templates`, `xsl:call-template`, `xsl:choose`, `xsl:fallback`, `xsl:for-each`, `xsl:for-each-group`, `xsl:if`, `xsl:matching-substring`, `xsl:next-match`, `xsl:non-matching-substring`, `xsl:otherwise`, `xsl:perform-sort`, `xsl:sequence`, or `xsl:when`
- The sequence may be used to construct the content of a new element or document node. This happens when the sequence constructor appears as the content of a literal result element, or of one of the instructions `xsl:copy`, `xsl:element`, `xsl:document`, `xsl:result-document`, or `xsl:message`. It also happens when the sequence constructor is contained in one of the elements `xsl:variable`, `xsl:param`, or `xsl:with-param`, when this instruction has no `as` attribute. For details, see *5.7.1 Constructing Complex Content*.
- The sequence may be used to construct the string value of an attribute node, text node, namespace node, comment node, or processing instruction node. This happens when the sequence constructor is contained in one of the elements `xsl:attribute`, `xsl:value-of`, `xsl:namespace`, `xsl:comment`, or `xsl:processing-instruction`. For details, see *5.7.2 Constructing Simple Content*.

Note:

The term *sequence constructor* replaces *template* as used in XSLT 1.0. The change is made partly for clarity (to avoid confusion with template rules and named templates), but also to reflect a more formal definition of the semantics. Whereas XSLT 1.0 described a template as a sequence of instructions that write to the result tree, XSLT 2.0 describes a sequence constructor as something that can be evaluated to return a sequence of items; what happens to these items depends on the containing instruction.

5.7.1 Constructing Complex Content

This section describes how the sequence obtained by evaluating a sequence constructor may be used to construct the children of a newly constructed document node, or the children, attributes and namespaces of a newly constructed element node. The sequence of items may be obtained by evaluating the sequence constructor contained in an instruction such as `xsl:copy`, `xsl:element`, `xsl:document`, `xsl:result-document`, or a literal result element.

When constructing the content of an element, the `inherit-namespaces` attribute of the `xsl:element` or `xsl:copy` instruction, or the `xsl:inherit-namespaces` property of the literal result element, determines whether namespace nodes are to be inherited. The effect of this attribute is described in the rules that follow.

The sequence is processed as follows (applying the rules in the order they are listed):

1. The containing instruction may generate attribute nodes and/or namespace nodes, as specified in the rules for the individual instruction. For example, these nodes may be produced by expanding an `[xsl:]use-attribute-sets` attribute, or by expanding the attributes of a literal result element. Any such nodes are prepended to the sequence produced by evaluating the sequence constructor.
2. Any atomic value in the sequence is cast to a string.

   Note:

   Casting from `xs:QName` or `xs:NOTATION` to `xs:string` always succeeds, because these values retain a prefix for this purpose. However, there is no guarantee that the prefix used will always be meaningful in the context where the resulting string is used.
3. Any consecutive sequence of strings within the result sequence is converted to a single text node, whose string value contains the content of each of the strings in turn, with a single space (#x20) used as a separator between successive strings.
4. Any document node within the result sequence is replaced by a sequence containing each of its children, in document order.
5. Zero-length text nodes within the result sequence are removed.
6. Adjacent text nodes within the result sequence are merged into a single text node.
7. Invalid namespace and attribute nodes are detected as follows.

   [ERR XTDE0410] It is a non-recoverable dynamic error if the result sequence used to construct the content of an element node contains a namespace node or attribute node that is preceded in the sequence by a node that is neither a namespace node nor an attribute node.

   [ERR XTDE0420] It is a non-recoverable dynamic error if the result sequence used to construct the content of a document node contains a namespace node or attribute node.

   [ERR XTDE0430] It is a non-recoverable dynamic error if the result sequence contains two or more namespace nodes having the same name but different string values (that is, namespace nodes that map the same prefix to different namespace URIs).

   [ERR XTDE0440] It is a non-recoverable dynamic error if the result sequence contains a namespace node with no name and the element node being constructed has a null namespace URI (that is, it is an error to define a default namespace when the element is in no

namespace).

8. If the result sequence contains two or more namespace nodes with the same name (or no name) and the same string value (that is, two namespace nodes mapping the same prefix to the same namespace URI), then all but one of the duplicate nodes are discarded.

   Note:

   Since the order of namespace nodes is undefined, it is not significant which of the duplicates is retained.

9. If an attribute *A* in the result sequence has the same name as another attribute *B* that appears later in the result sequence, then attribute *A* is discarded from the result sequence.
10. Each node in the resulting sequence is attached as a namespace, attribute, or child of the newly constructed element or document node. Conceptually this involves making a deep copy of the node; in practice, however, copying the node will only be necessary if the existing node can be referenced independently of the parent to which it is being attached. When copying an element or processing instruction node, its base URI property is changed to be the same as that of its new parent, unless it has an `xml:base` attribute (see [XML Base]) that overrides this. If the copied element has an `xml:base` attribute, its base URI is the value of that attribute, resolved (if it is relative) against the base URI of the new parent node.
11. If the newly constructed node is an element node, then namespace fixup is applied to this node, as described in *5.7.3 Namespace Fixup*.
12. If the newly constructed node is an element node, and if namespaces are inherited, then each namespace node of the newly constructed element (including any produced as a result of the namespace fixup process) is copied to each descendant element of the newly constructed element, unless that element or an intermediate element already has a namespace node with the same name (or absence of a name) or that descendant element or an intermediate element is in no namespace and the namespace node has no name.

Example: A Sequence Constructor for Complex Content

Consider the following stylesheet fragment:

```
<td>
  <xsl:attribute name="valign">top</xsl:attribute>
  <xsl:value-of select="@description"/>
</td>
```

This fragment consists of a literal result element `td`, containing a sequence constructor that consists of two instructions: `xsl:attribute` and `xsl:value-of`. The sequence constructor is evaluated to produce a sequence of two nodes: a parentless attribute node, and a parentless text node. The `td` instruction causes a `td` element to be created; the new attribute therefore becomes an attribute of the new `td` element, while the text node created by the `xsl:value-of` instruction becomes a child of the `td` element (unless it is zero-length, in which case it is discarded).

Example: Space Separators in Element Content

Consider the following stylesheet fragment:

```
<doc>
  <e><xsl:sequence select="1 to 5"/></e>
  <f>
    <xsl:for-each select="1 to 5">
      <xsl:value-of select="."/>
    </xsl:for-each>
  </f>
</doc>
```

This produces the output (when indented):

```
<doc>
  <e>1 2 3 4 5</e>
  <f>12345</f>
</doc>
```

The difference between the two cases is that for the `e` element, the sequence constructor generates a sequence of five atomic values, which are therefore separated by spaces. For the `f` element, the content is a sequence of five text nodes, which are concatenated without space separation.

It is important to be aware of the distinction between `xsl:sequence`, which returns the value of its `select` expression unchanged, and `xsl:value-of`, which constructs a text node.

5.7.2 Constructing Simple Content

The `xsl:attribute`, `xsl:comment`, `xsl:processing-instruction`, `xsl:namespace`, and `xsl:value-of` elements create nodes that cannot have children. Specifically, the `xsl:attribute` instruction creates an attribute node, `xsl:comment` creates a comment node, `xsl:processing-instruction` creates a processing instruction node, `xsl:namespace` creates a namespace node, and `xsl:value-of` creates a text node. The string value of the new node is constructed using either the `select` attribute of the instruction, or the sequence constructor that forms the content of the instruction. The `select` attribute allows the content to be specified by means of an XPath expression, while the sequence constructor allows it to be specified by means of a sequence of XSLT instructions. The `select` attribute or sequence constructor is evaluated to produce a result sequence, and the string value of the new node is derived from this result sequence according to the rules below.

These rules are also used to compute the effective value of an attribute value template. In this case the sequence being processed is the result of evaluating an XPath expression enclosed between curly brackets, and the separator is a single space character.

1. Zero-length text nodes in the sequence are discarded.
2. Adjacent text nodes in the sequence are merged into a single text node.
3. The sequence is atomized.
4. Every value in the atomized sequence is cast to a string.
5. The strings within the resulting sequence are concatenated, with a (possibly zero-length) separator inserted between successive strings. The default separator is a single space. In the case of `xsl:attribute` and `xsl:value-of`, a different separator can be specified using the `separator` attribute of the instruction; it is permissible for this to be a zero-length string, in which case the strings are concatenated with no separator. In the case of `xsl:comment`, `xsl:processing-instruction`, and `xsl:namespace` , and when expanding an attribute value template, the default separator cannot be changed.
6. In the case of `xsl:processing-instruction`, any leading spaces in the resulting string are removed.
7. The resulting string forms the string value of the new attribute, namespace, comment, processing-instruction, or text node.

Example: Space Separators in Attribute Content

Consider the following stylesheet fragment:

```
<doc>
  <xsl:attribute name="e" select="1 to 5"/>
  <xsl:attribute name="f">
    <xsl:for-each select="1 to 5">
      <xsl:value-of select="."/>
    </xsl:for-each>
  </xsl:attribute>
</doc>
```

This produces the output:

```
<doc e="1 2 3 4 5" f="12345"/>
```

The difference between the two cases is that for the `e` attribute, the sequence constructor generates a sequence of five atomic values, which are therefore separated by spaces. For the `f` attribute, the content is supplied as a sequence of five text nodes, which are concatenated without space separation.

Specifying `separator=""` on the first `xsl:attribute` instruction would cause the attribute value to be `e="12345"`. A `separator` attribute on the second `xsl:attribute` instruction would have no effect, since the separator only affects the way adjacent atomic values are handled: separators are never inserted between adjacent text nodes.

Note:

If an attribute value template contains a sequence of fixed and variable parts, no additional whitespace is inserted between the expansions of the fixed and variable parts. For example, the effective value of the attribute `a="chapters{4 to 6}"` is `a="chapters4 5 6"`.

5.7.3 Namespace Fixup

In a tree supplied to or constructed by an XSLT processor, the constraints relating to namespace nodes that are specified in [Data Model] MUST be satisfied. For example

- If an element node has an expanded-QName with a non-null namespace URI, then that element node MUST have at least one namespace node whose string value is the same as that namespace URI.
- If an element node has an attribute node whose expanded-QName has a non-null namespace URI, then the element MUST have at least one namespace node whose string value is the same as that namespace URI and whose name is non-empty.
- Every element MUST have a namespace node whose expanded-QName has local-part `xml` and whose string value is `http://www.w3.org/XML/1998/namespace`. The namespace prefix `xml` must not be associated with any other namespace URI, and the namespace URI `http://www.w3.org/XML/1998/namespace` must not be associated with any other prefix.
- A namespace node MUST NOT have the name `xmlns`.

[DEFINITION: The rules for the individual XSLT instructions that construct a result tree (see *11 Creating Nodes and Sequences*) prescribe some of the situations in which namespace nodes are written to the tree. These rules, however, are not sufficient to ensure that the prescribed constraints are always satisfied. The XSLT processor MUST therefore add additional namespace nodes to satisfy these constraints. This process is referred to as **namespace fixup**.]

The actual namespace nodes that are added to the tree by the namespace fixup process are implementation-dependent, provided firstly, that at the end of the process the above constraints MUST all be satisfied, and secondly, that a namespace node MUST NOT be added to the tree unless the namespace node is necessary either to satisfy these constraints, or to enable the tree to be serialized using the original namespace prefixes from the source document or stylesheet.

Namespace fixup MUST NOT result in an element having multiple namespace nodes with the same name.

Namespace fixup MAY, if necessary to resolve conflicts, change the namespace prefix contained in the QName value that holds the name of an element or attribute node. This includes the option to add or remove a prefix. However, namespace fixup MUST NOT change the prefix component contained in a value of type `xs:QName` or `xs:NOTATION` that forms the typed value of an element or attribute node.

Note:

Namespace fixup is not used to create namespace declarations for `xs:QName` or `xs:NOTATION` values appearing in the content of an element or attribute.

Where values acquire such types as the result of validation, namespace fixup does not come into play, because namespace fixup happens before validation: in this situation, it is the user's responsibility to ensure that the element being validated has the required

namespace nodes to enable validation to succeed.

Where existing elements are copied along with their existing type annotations (`validation="preserve"`) the rules require that existing namespace nodes are also copied, so that any namespace-sensitive values remain valid.

Where existing attributes are copied along with their existing type annotations, the rules of the XDM data model require that a parentless attribute node cannot contain a namespace-sensitive typed value; this means that it is an error to copy an attribute using `validation="preserve"` if it contains namespace-sensitive content.

[ERR XTDE0485] It is a non-recoverable dynamic error if namespace fixup is performed on an element that contains among the typed values of the element and its attributes two values of type `xs:QName` or `xs:NOTATION` containing conflicting namespace prefixes, that is, two values that use the same prefix to refer to different namespace URIs.

Namespace fixup is applied to every element that is constructed using a literal result element, or one of the instructions `xsl:element`, `xsl:copy`, or `xsl:copy-of`. An implementation is not REQUIRED to perform namespace fixup for elements in any source document, that is, for a document in the initial input sequence, documents loaded using the `document`, `doc`[FO] or `collection`[FO] function, documents supplied as the value of a stylesheet parameter, or documents returned by an extension function or extension instruction.

Note:

A source document (an input document, a document returned by the `document`, `doc`[FO] or `collection`[FO] functions, a document returned by an extension function or extension instruction, or a document supplied as a stylesheet parameter) is required to satisfy the constraints described in [Data Model], including the constraints imposed by the namespace fixup process. The effect of supplying a pseudo-document that does not meet these constraints is undefined.

In an Infoset (see [XML Information Set]) created from a document conforming to [Namespaces in XML 1.0], it will always be true that if a parent element has an in-scope namespace with a non-empty namespace prefix, then its child elements will also have an in-scope namespace with the same namespace prefix, though possibly with a different namespace URI. This constraint is removed in [Namespaces in XML 1.1]. XSLT 2.0 supports the creation of result trees that do not satisfy this constraint: the namespace fixup process does not add a namespace node to an element merely because its parent node in the result tree has such a namespace node. However, the process of constructing the children of a new element, which is described in *5.7.1 Constructing Complex Content*, does cause the namespaces of a parent element to be inherited by its children unless this is prevented using `[xsl:]inherit-namespaces="no"` on the instruction that creates the parent element.

Note:

This has implications on serialization, defined in [XSLT and XQuery Serialization]. It means that it is possible to create final result trees that cannot be faithfully serialized as XML 1.0 documents. When such a result tree is serialized as XML 1.0, namespace declarations written for the parent element will be inherited by its child elements as if the corresponding namespace nodes were present on the child element, except in the case of the default namespace, which can be undeclared using the construct `xmlns=""`. When the same result tree is serialized as XML 1.1, however, it is possible to undeclare any namespace on the child element (for example, `xmlms:foo=""`) to prevent this inheritance taking place.

5.8 URI References

[DEFINITION: Within this specification, the term **URI Reference**, unless otherwise stated, refers to a string in the lexical space of the `xs:anyURI` data type as defined in [XML Schema Part 2].] Note that this is a wider definition than that in [RFC3986]: in particular, it is designed to accommodate Internationalized Resource Identifiers (IRIs) as described in [RFC3987], and thus allows the use of non-ASCII characters without escaping.

URI References are used in XSLT with three main roles:

As namespace URIs
As collation URIs
As identifiers for resources such as stylesheet modules; these resources are typically accessible using a protocol such as HTTP. Examples of such identifiers are the URIs used in the `href` attributes of `xsl:import`, `xsl:include`, and `xsl:result-document`.

The rules for namespace URIs are given in [Namespaces in XML 1.0] and [Namespaces in XML 1.1]. Those specifications deprecate the use of relative URIs as namespace URIs.

The rules for collation URIs are given in [Functions and Operators].

URI references used to identify external resources must conform to the same rules as the locator attribute (`href`) defined in section 5.4 of [XLink]. If the URI reference is relative, then it is resolved (unless otherwise specified) against the base URI of the containing element node, according to the rules of [RFC3986], after first escaping all characters that need to be escaped to make it a valid RFC3986 URI reference. (But a relative URI in the `href` attribute of `xsl:result-document` is resolved against the Base Output URI.)

Other URI references appearing in an XSLT stylesheet document, for example the system identifiers of external entities or the value of the `xml:base` attribute, must follow the rules in their respective specifications.

6 Template Rules

Template rules define the processing that can be applied to nodes that match a particular pattern.

6.1 Defining Templates

```
<!-- Category: declaration -->
<xsl:template
  match? = pattern
  name? = qname
  priority? = number
  mode? = tokens
  as? = sequence-type>
  <!-- Content: (xsl:param*, sequence-constructor) -->
</xsl:template>
```

[DEFINITION: An `xsl:template` declaration defines a **template**, which contains a sequence constructor for creating nodes and/or atomic values. A template can serve either as a template rule, invoked by matching nodes against a pattern, or as a named template, invoked explicitly by name. It is also possible for the same template to serve in both capacities.]

[ERR XTSE0500] An `xsl:template` element MUST have either a `match` attribute or a `name` attribute, or both. An `xsl:template` element that has no `match` attribute MUST have no `mode` attribute and no `priority` attribute.

If an `xsl:template` element has a `match` attribute, then it is a template rule. If it has a `name` attribute, then it is a named template.

A template may be invoked in a number of ways, depending on whether it is a template rule, a named template, or both. The result of invoking the template is the result of evaluating the sequence constructor contained in the `xsl:template` element (see *5.7 Sequence Constructors*).

If an `as` attribute is present, the `as` attribute defines the required type of the result. The result of evaluating the sequence constructor is then converted to the required type using the function conversion rules. If no `as` attribute is specified, the default value is `item()*`, which permits any value. No conversion then takes place.

[ERR XTTE0505] It is a type error if the result of evaluating the sequence constructor cannot be converted to the required type.

6.2 Defining Template Rules

This section describes template rules. Named templates are described in *10.1 Named Templates*.

A template rule is specified using the `xsl:template` element with a `match` attribute. The `match` attribute is a Pattern that identifies the node or nodes to which the rule applies. The result of applying the template rule is the result of evaluating the sequence constructor contained in the `xsl:template` element, with the matching node used as the context node.

Example: A simple Template Rule

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:wrapper` element with a `font-weight` property of `bold`.

```
<xsl:template match="emph">
  <fo:wrapper font-weight="bold" xmlns:fo="http://www.w3.org/1999/XSL/Format">
    <xsl:apply-templates/>
  </fo:wrapper>
</xsl:template>
```

A template rule is evaluated when an `xsl:apply-templates` instruction selects a node that matches the pattern specified in the `match` attribute. The `xsl:apply-templates` instruction is described in the next section. If several template rules match a selected node, only one of them is evaluated, as described in *6.4 Conflict Resolution for Template Rules*.

6.3 Applying Template Rules

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

The `xsl:apply-templates` instruction takes as input a sequence of nodes (typically nodes in a source tree), and produces as output a sequence of items; these will often be nodes to be added to a result tree.

If the instruction has one or more `xsl:sort` children, then the input sequence is sorted as described in *13 Sorting*. The result of this sort is referred to below as the **sorted sequence**; if there are no `xsl:sort` elements, then the sorted sequence is the same as the input sequence.

Each node in the input sequence is processed by finding a template rule whose pattern matches that node. If there is more than one, the best among them is chosen, using rules described in *6.4 Conflict Resolution for Template Rules*. If there is no template rule whose pattern matches the node, a built-in template rule is used (see *6.6 Built-in Template Rules*). The chosen template rule is evaluated. The rule that matches the *N*th node in the sorted sequence is evaluated with that node as the context item, with *N* as the context position, and with the length of the sorted sequence as the context size. Each template rule that is evaluated produces a sequence of items as its result. The resulting sequences (one for each node in the sorted sequence) are then concatenated, to form a single sequence. They are concatenated retaining the order of the nodes in the sorted sequence. The final concatenated sequence forms the result of the `xsl:apply-templates` instruction.

Example: Applying Template Rules

Suppose the source document is as follows:

```
<message>Proceed <emph>at once</emph> to the exit!</message>
```

This can be processed using the two template rules shown below.

```
<xsl:template match="message">
  <p>
    <xsl:apply-templates select="child::node()"/>
```

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

<xsl:template match="emph">
  <b>
    <xsl:apply-templates select="child::node()"/>
  </b>
</xsl:template>
```

There is no template rule for the document node; the built-in template rule for this node will cause the `message` element to be processed. The template rule for the `message` element causes a `p` element to be written to the result tree; the contents of this `p` element are constructed as the result of the `xsl:apply-templates` instruction. This instruction selects the three child nodes of the `message` element (a text node containing the value "`Proceed `", an `emph` element node, and a text node containing the value "` to the exit!`"). The two text nodes are processed using the built-in template rule for text nodes, which returns a copy of the text node. The `emph` element is processed using the explicit template rule that specifies `match="emph"`.

When the `emph` element is processed, this template rule constructs a `b` element. The contents of the `b` element are constructed by means of another `xsl:apply-templates` instruction, which in this case selects a single node (the text node containing the value "`at once`"). This is again processed using the built-in template rule for text nodes, which returns a copy of the text node.

The final result of the `match="message"` template rule thus consists of a `p` element node with three children: a text node containing the value "`Proceed `", a `b` element that is the parent of a text node containing the value "`at once`", and a text node containing the value "` to the exit!`". This result tree might be serialized as:

```
<p>Proceed <b>at once</b> to the exit!</p>
```

The default value of the `select` attribute is `child::node()`, which causes all the children of context node to be processed.

[ERR XTTE0510] It is a type error if an `xsl:apply-templates` instruction with no `select` attribute is evaluated when the context item is not a node.

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 sequence of nodes (it can contain zero, one, or more nodes).

[ERR XTTE0520] It is a type error if the sequence returned by the `select` expression contains an item that is not a node.

Note:

In XSLT 1.0, the `select` attribute selected a set of nodes, which by default were processed in document order. In XSLT 2.0, it selects a sequence of nodes. In cases that would have been valid in XSLT 1.0, the expression will return a sequence of nodes in document order, so the effect is the same.

Example: Applying Templates to Selected Nodes

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

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

Example: Applying Templates to Nodes that are not Descendants

It is also possible to process elements that are not descendants of the context 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>
```

Example: Matching by Schema-Defined Types

It is possible to write template rules that are matched according to the schema-defined type of an element or attribute. The following example applies different formatting to the children of an element depending on their type:

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

```
<xsl:template match="product/*" priority="3">
  <tr>
    <td><xsl:value-of select="name()"/></td>
    <td><xsl:next-match/></td>
  </tr>
</xsl:template>

<xsl:template match="product/element(*, xs:decimal) |
                     product/element(*, xs:double)" priority="2">
  <xsl:value-of select="format-number(xs:double(.), '#,###0.00')"/>
</xsl:template>

<xsl:template match="product/element(*, xs:date)" priority="2">
  <xsl:value-of select="format-date(., '[Mn] [D], [Y]')"/>
</xsl:template>

<xsl:template match="product/*" priority="1.5">
  <xsl:value-of select="."/>
</xsl:template>
```

The xsl:next-match instruction is described in *6.7 Overriding Template Rules*.

Example: Re-ordering Elements in the Result Tree

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>
```

Example: Processing Recursive Structures

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><div><div></div></div></doc>
```

the rule

```
<xsl:template match="doc">
  <xsl:apply-templates select=".//div"/>
</xsl:template>
```

will process both the outer `div` and inner `div` elements.

This means that if the template rule for the `div` element processes its own children, then these grandchildren will be processed more than once, which is probably not what is required. The solution is to process one level at a time in a recursive descent, by using `select="div"` in place of `select=".//div"`

Note:

The xsl:apply-templates instruction is most commonly used to process nodes that are descendants of the context 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 context 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.

6.4 Conflict Resolution for Template Rules

It is possible for a node in a source document to match more than one template rule. When this happens, only one template rule is evaluated for the node. The template rule to be used is determined as follows:

1. First, only the matching template rule or rules with the highest import precedence are considered. Other matching template rules with lower precedence are eliminated from consideration.
2. Next, of the remaining matching rules, only those with the highest priority are considered. Other matching template rules with lower priority are eliminated from consideration. The priority of a template rule is specified by the `priority` attribute on the `xsl:template` declaration.

   [ERR XTSE0530] The value of this attribute MUST conform to the rules for the `xs:decimal` type defined in [XML Schema Part 2]. Negative values are permitted..

   [DEFINITION: If no `priority` attribute is specified on the `xsl:template` element, a **default priority** is computed, based on the syntax of the pattern supplied in the `match` attribute.] The rules are as follows:

   - If the pattern contains multiple alternatives separated by `|`, then the template rule is treated equivalently to a set of template rules, one for each alternative. However, it is not an error if a node matches more than one of the alternatives.
   - If the pattern has the form `/`, then the priority is −0.5.
   - If the pattern has the form of a QName optionally preceded by a PatternAxis or has the form `processing-instruction(` StringLiteral[XP] `)` or `processing-instruction(`NCName[Names]`)` optionally preceded by a PatternAxis, then the priority is 0.
   - If the pattern has the form of an ElementTest[XP] or AttributeTest[XP], optionally preceded by a PatternAxis, then the priority is as shown in the table below. In this table, the symbols *E*, *A*, and *T* represent an arbitrary element name, attribute name, and type name respectively, while the symbol `*` represents itself. The presence or absence of the symbol `?` following a type name does not affect the priority.

Format	Priority	Notes
`element()`	−0.5	(equivalent to `*`)
`element(*)`	−0.5	(equivalent to `*`)
`attribute()`	−0.5	(equivalent to `@*`)
`attribute(*)`	−0.5	(equivalent to `@*`)
`element(`*E*`)`	0	(equivalent to E)
`element(*,`*T*`)`	0	(matches by type only)
`attribute(`*A*`)`	0	(equivalent to `@`A)
`attribute(*,`*T*`)`	0	(matches by type only)
`element(`*E*`,`*T*`)`	0.25	(matches by name and type)
`schema-element(`*E*`)`	0.25	(matches by substitution group and type)
`attribute(`*A*`,`*T*`)`	0.25	(matches by name and type)
`schema-attribute(`*A*`)`	0.25	(matches by name and type)

   - If the pattern has the form of a DocumentTest[XP], then if it includes no ElementTest[XP] or SchemaElementTest[XP] the priority is −0.5. If it does include an ElementTest[XP] or SchemaElementTest[XP], then the priority is the same as the priority of that ElementTest[XP] or SchemaElementTest[XP], computed according to the table above.
   - If the pattern has the form NCName[Names]`:*` or `*:`NCName[Names], optionally preceded by a PatternAxis, then the priority is −0.25.
   - If the pattern is any other NodeTest[XP], optionally preceded by a PatternAxis, then the priority is −0.5.
   - Otherwise, the priority is 0.5.

   Note:

   In many cases this means that highly selective patterns have higher priority than less selective patterns. The most common kind of pattern (a pattern that tests for a node of a particular kind, with a particular expanded-QName or a particular type) has priority 0. The next less specific kind of pattern (a pattern that tests for a node of a particular kind and an expanded-QName with a particular namespace URI) has priority −0.25. Patterns less specific than this (patterns that just test for nodes of a given kind) have priority −0.5. Patterns that specify both the name and the required type have a priority of +0.25, putting them above patterns that only specify the name *or* the type. Patterns more specific than this, for example patterns that include predicates or that specify the ancestry of the required node, have priority 0.5.

   However, it is not invariably true that a more selective pattern has higher priority than a less selective pattern. For example, the priority of the pattern `node()[self::*]` is higher than that of the pattern `salary`. Similarly, the patterns `attribute(*, xs:decimal)` and `attribute(*, xs:short)` have the same priority, despite the fact that the latter pattern matches a subset of the nodes matched by the former. Therefore, to achieve clarity in a stylesheet it is good practice to allocate explicit priorities.

[ERR XTRE0540] It is a recoverable dynamic error if the conflict resolution algorithm for template rules leaves more than one matching template rule. The optional recovery action is to select, from the matching template rules that are left, the one that occurs last in declaration order.

6.5 Modes

[DEFINITION: **Modes** allow a node in a source tree to be processed multiple times, each time producing a different result. They also allow different sets of template rules to be active when processing different trees, for example when processing documents loaded using the `document` function (see *16.1 Multiple Source Documents*) or when processing temporary trees.]

[DEFINITION: There is always a **default mode** available. The default mode is an unnamed mode, and it is used when no `mode` attribute is specified on an `xsl:apply-templates` instruction.]

Every mode other than the default mode is identified by a QName.

A template rule is applicable to one or more modes. The modes to which it is applicable are defined by the `mode` attribute of the `xsl:template` element. If the attribute is omitted, then the template rule is applicable to the default mode. If the attribute is present, then its value MUST be a non-empty whitespace-separated list of tokens, each of which defines a mode to which the template rule is applicable. Each token MUST be one of the following:

- a QName, which is expanded as described in *5.1 Qualified Names* to define the name of the mode
- the token `#default`, to indicate that the template rule is applicable to the default mode
- the token `#all`, to indicate that the template rule is applicable to all modes (that is, to the default mode and to every mode that is named in an `xsl:apply-templates` instruction or `xsl:template` declaration anywhere in the stylesheet).

[ERR XTSE0550] It is a static error if the list is empty, if the same token is included more than once in the list, if the list contains an invalid token, or if the token `#all` appears together with any other value.

The `xsl:apply-templates` element also has an optional `mode` attribute. The value of this attribute MUST either be a QName, which is expanded as described in *5.1 Qualified Names* to define the name of a mode, or the token `#default`, to indicate that the default mode is to be used, or the token `#current`, to indicate that the current mode is to be used. If the attribute is omitted, the default mode is used.

When searching for a template rule to process each node selected by the `xsl:apply-templates` instruction, only those template rules that are applicable to the selected mode are considered.

[DEFINITION: At any point in the processing of a stylesheet, there is a **current mode**. When the transformation is initiated, the current mode is the default mode, unless a different initial mode has been supplied, as described in *2.3 Initiating a Transformation*. Whenever an `xsl:apply-templates` instruction is evaluated, the current mode becomes the mode selected by this instruction.] When a stylesheet function is called, the current mode becomes the default mode. No other instruction changes the current mode. On completion of the `xsl:apply-templates` instruction, or on return from a stylesheet function call, the current mode reverts to its previous value. The current mode is used when an `xsl:apply-templates` instruction uses the syntax `mode="#current"`; it is also used by the `xsl:apply-imports` and `xsl:next-match` instructions (see *6.7 Overriding Template Rules*).

6.6 Built-in Template Rules

When a node is selected by `xsl:apply-templates` and there is no template rule in the stylesheet that can be used to process that node, a built-in template rule is evaluated instead.

The built-in template rules apply to all modes.

The built-in rule for document nodes and element nodes is equivalent to calling `xsl:apply-templates` with no `select` attribute, and with the `mode` attribute set to `#current`. If the built-in rule was invoked with parameters, those parameters are passed on in the implicit `xsl:apply-templates` instruction.

Example: Using a Built-In Template Rule

For example, suppose the stylesheet contains the following instruction:

```
<xsl:apply-templates select="title" mode="mm">
  <xsl:with-param name="init" select="10"/>
</xsl:apply-templates>
```

If there is no explicit template rule that matches the `title` element, then the following implicit rule is used:

```
<xsl:template match="title" mode="#all">
  <xsl:param name="init"/>
  <xsl:apply-templates mode="#current">
    <xsl:with-param name="init" select="$init"/>
  </xsl:apply-templates>
</xsl:template>
```

The built-in template rule for text and attribute nodes returns a text node containing the string value of the context node. It is effectively:

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

Note:

This text node may have a string value that is zero-length.

The built-in template rule for processing instructions and comments does nothing (it returns the empty sequence).

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

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 always used when `xsl:apply-templates` selects a namespace node.

The built-in template rules have lower import precedence than all other template rules. Thus, the stylesheet author can override a built-in template rule by including an explicit template rule.

6.7 Overriding Template Rules

```
<!-- Category: instruction -->
<xsl:apply-imports>
  <!-- Content: xsl:with-param* -->
</xsl:apply-imports>
```

```
<!-- Category: instruction -->
<xsl:next-match>
  <!-- Content: (xsl:with-param | xsl:fallback)* -->
```

```
</xsl:next-match>
```

A template rule that is being used to override another template rule (see *6.4 Conflict Resolution for Template Rules*) can use the `xsl:apply-imports` or `xsl:next-match` instruction to invoke the overridden template rule. The `xsl:apply-imports` instruction only considers template rules in imported stylesheet modules; the `xsl:next-match` instruction considers all other template rules of lower import precedence and/or priority. Both instructions will invoke the built-in template rule for the node (see *6.6 Built-in Template Rules*) if no other template rule is found.

[DEFINITION: At any point in the processing of a stylesheet, there may be a **current template rule**. Whenever a template rule is chosen as a result of evaluating `xsl:apply-templates`, `xsl:apply-imports`, or `xsl:next-match`, the template rule becomes the current template rule for the evaluation of the rule's sequence constructor. When an `xsl:for-each`, `xsl:for-each-group`, or `xsl:analyze-string` instruction is evaluated, or when evaluating a sequence constructor contained in an `xsl:sort` or `xsl:key` element, or when a stylesheet function is called (see *10.3 Stylesheet Functions*), the current template rule becomes null for the evaluation of that instruction or function.]

The current template rule is not affected by invoking named templates (see *10.1 Named Templates*) or named attribute sets (see *10.2 Named Attribute Sets*). While evaluating a global variable or the default value of a stylesheet parameter (see *9.5 Global Variables and Parameters*) the current template rule is null.

Note:

These rules ensure that when `xsl:apply-imports` or `xsl:next-match` is called, the context item is the same as when the current template rule was invoked, and is always a node.

Both `xsl:apply-imports` and `xsl:next-match` search for a template rule that matches the context node, and that is applicable to the current mode (see *6.5 Modes*). In choosing a template rule, they use the usual criteria such as the priority and import precedence of the template rules, but they consider as candidates only a subset of the template rules in the stylesheet. This subset differs between the two instructions:

- The `xsl:apply-imports` instruction considers as candidates only those template rules contained in stylesheet levels that are descendants in the import tree of the stylesheet level that contains the current template rule.

  Note:

  This is *not* the same as saying that the search considers all template rules whose import precedence is lower than that of the current template rule.
- The `xsl:next-match` instruction considers as candidates all those template rules that come after the current template rule in the ordering of template rules implied by the conflict resolution rules given in *6.4 Conflict Resolution for Template Rules*. That is, it considers all template rules with lower import precedence than the current template rule, plus the template rules that are at the same import precedence that have lower priority than the current template rule. If the processor has recovered from the error that occurs when two matching template rules have the same import precedence and priority, then it also considers all matching template rules with the same import precedence and priority that occur before the current template rule in declaration order.

  Note:

  As explained in *6.4 Conflict Resolution for Template Rules*, a template rule whose match pattern contains multiple alternatives separated by `|` is treated equivalently to a set of template rules, one for each alternative. This means that where the same node matches more than one alternative, and the alternatives have different priority, it is possible for an `xsl:next-match` instruction to cause the current template rule to be invoked recursively. This situation does not occur when the alternatives have the same priority.

If no matching template rule is found that satisfies these criteria, the built-in template rule for the node kind is used (see *6.6 Built-in Template Rules*).

An `xsl:apply-imports` or `xsl:next-match` instruction may use `xsl:with-param` child elements to pass parameters to the chosen template rule (see *10.1.1 Passing Parameters to Templates*). It also passes on any tunnel parameters as described in *10.1.2 Tunnel Parameters*.

[ERR XTDE0560] It is a non-recoverable dynamic error if `xsl:apply-imports` or `xsl:next-match` is evaluated when the current template rule is null.

Example: Using `xsl:apply-imports`

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>
```

An `xsl:fallback` instruction appearing as a child of an `xsl:next-match` instruction is ignored by an XSLT 2.0 processor, but can be used to define fallback behavior when the stylesheet is processed by an XSLT 1.0 processor in forwards-compatible mode.

7 Repetition

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

The xsl:for-each instruction processes each item in a sequence of items, evaluating the sequence constructor within the xsl:for-each instruction once for each item in that sequence.

The select attribute is REQUIRED, and the expression MUST evaluate to a sequence, called the input sequence. If there is an xsl:sort element present (see *13 Sorting*) the input sequence is sorted to produce a sorted sequence. Otherwise, the sorted sequence is the same as the input sequence.

The xsl:for-each instruction contains a sequence constructor. The sequence constructor is evaluated once for each item in the sorted sequence, with the focus set as follows:

- The context item is the item being processed. If this is a node, it will also be the context node. If it is not a node, there will be no context node: that is, any attempt to reference the context node will result in a non-recoverable dynamic error.
- The context position is the position of this item in the sorted sequence.
- The context size is the size of the sorted sequence (which is the same as the size of the input sequence).

For each item in the input sequence, evaluating the sequence constructor produces a sequence of items (see *5.7 Sequence Constructors*). These output sequences are concatenated; if item *Q* follows item *P* in the sorted sequence, then the result of evaluating the sequence constructor with *Q* as the context item is concatenated after the result of evaluating the sequence constructor with *P* as the context item. The result of the xsl:for-each instruction is the concatenated sequence of items.

Note:

With XSLT 1.0, the selected nodes were processed in document order. With XSLT 2.0, XPath expressions that would have been valid under XPath 1.0 (such as path expressions and union expressions) will return a sequence of nodes that is already in document order, so backwards compatibility is maintained.

Example: Using `xsl:for-each`

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>
```

8 Conditional Processing

There are two instructions in XSLT that support conditional processing: 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.

8.1 Conditional Processing with xsl:if

```
<!-- Category: instruction -->
<xsl:if
  test = expression>
```

```
  <!-- Content: sequence-constructor -->
</xsl:if>
```

The xsl:if element has a mandatory test attribute, which specifies an expression. The content is a sequence constructor.

The result of the xsl:if instruction depends on the effective boolean value[XP] of the expression in the test attribute. The rules for determining the effective boolean value of an expression are given in [XPath 2.0]: they are the same as the rules used for XPath conditional expressions.

If the effective boolean value of the expression is true, then the sequence constructor is evaluated (see *5.7 Sequence Constructors*), and the resulting node sequence is returned as the result of the xsl:if instruction; otherwise, the sequence constructor is not evaluated, and the empty sequence is returned.

Example: Using `xsl:if`

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>
```

8.2 Conditional Processing with xsl:choose

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

```
<xsl:when
  test = expression>
  <!-- Content: sequence-constructor -->
</xsl:when>
```

```
<xsl:otherwise>
  <!-- Content: sequence-constructor -->
</xsl:otherwise>
```

The xsl:choose element selects one among a number of possible alternatives. It consists of a sequence of one or more 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 sequence constructor.

When an xsl:choose element is processed, each of the xsl:when elements is tested in turn (that is, in the order that the elements appear in the stylesheet), until one of the xsl:when elements is satisfied. If none of the xsl:when elements is satisfied, then the xsl:otherwise element is considered, as described below.

An xsl:when element is satisfied if the effective boolean value[XP] of the expression in its test attribute is true. The rules for determining the effective boolean value of an expression are given in [XPath 2.0]: they are the same as the rules used for XPath conditional expressions.

The content of the first, and only the first, xsl:when element that is satisfied is evaluated, and the resulting sequence is returned as the result of the xsl:choose instruction. If no xsl:when element is satisfied, the content of the xsl:otherwise element is evaluated, and the resulting sequence is returned as the result of the xsl:choose instruction. If no xsl:when element is satisfied, and no xsl:otherwise element is present, the result of the xsl:choose instruction is an empty sequence.

Only the sequence constructor of the selected xsl:when or xsl:otherwise instruction is evaluated. The test expressions for xsl:when instructions after the selected one are not evaluated.

Example: Using `xsl:choose`

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>
```

9 Variables and Parameters

[DEFINITION: The two elements `xsl:variable` and `xsl:param` are referred to as **variable-binding elements**].

[DEFINITION: The `xsl:variable` element declares a **variable**, which may be a global variable or a local variable.]

[DEFINITION: The `xsl:param` element declares a **parameter**, which may be a stylesheet parameter, a template parameter, or a function parameter. A parameter is a variable with the additional property that its value can be set by the caller when the stylesheet, the template, or the function is invoked.]

[DEFINITION: A variable is a binding between a name and a value. The **value** of a variable is any sequence (of nodes and/or atomic values), as defined in [Data Model].]

9.1 Variables

```
<!-- Category: declaration -->
<!-- Category: instruction -->
<xsl:variable
  name = qname
  select? = expression
  as? = sequence-type>
  <!-- Content: sequence-constructor -->
</xsl:variable>
```

The `xsl:variable` element has 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 *5.1 Qualified Names*.

The `xsl:variable` element has an optional `as` attribute, which specifies the required type of the variable. The value of the `as` attribute is a SequenceType [XP], as defined in [XPath 2.0].

[DEFINITION: The value of the variable is computed using the expression given in the `select` attribute or the contained sequence constructor, as described in *9.3 Values of Variables and Parameters*. This value is referred to as the **supplied value** of the variable.] If the `xsl:variable` element has a `select` attribute, then the sequence constructor MUST be empty.

If the `as` attribute is specified, then the supplied value of the variable is converted to the required type, using the function conversion rules.

[ERR XTTE0570] It is a type error if the supplied value of a variable cannot be converted to the required type.

If the `as` attribute is omitted, the supplied value of the variable is used directly, and no conversion takes place.

9.2 Parameters

```
<!-- Category: declaration -->
<xsl:param
  name = qname
  select? = expression
  as? = sequence-type
  required? = "yes" | "no"
  tunnel? = "yes" | "no">
  <!-- Content: sequence-constructor -->
</xsl:param>
```

The `xsl:param` element may be used as a child of `xsl:stylesheet`, to define a parameter to the transformation; or as a child of `xsl:template` to define a parameter to a template, which may be supplied when the template is invoked using `xsl:call-template`, `xsl:apply-templates`, `xsl:apply-imports` or `xsl:next-match`; or as a child of `xsl:function` to define a parameter to a stylesheet function, which may be supplied when the function is called from an XPath expression.

The `xsl:param` element has a REQUIRED `name` attribute, which specifies the name of the parameter. The value of the `name` attribute is a QName, which is expanded as described in *5.1 Qualified Names*.

[ERR XTSE0580] It is a static error if two parameters of a template or of a stylesheet function have the same name.

> **Note:**
>
> For rules concerning stylesheet parameters, see *9.5 Global Variables and Parameters*. Local variables may shadow template parameters and function parameters: see *9.7 Scope of Variables*.

The supplied value of the parameter is the value supplied by the caller. If no value was supplied by the caller, and if the parameter is not mandatory, then the supplied value is computed using the expression given in the `select` attribute or the contained sequence constructor, as described in *9.3 Values of Variables and Parameters*. If the `xsl:param` element has a `select` attribute, then the sequence constructor MUST be empty.

> **Note:**
>
> This specification does not dictate whether and when the default value of a parameter is evaluated. For example, if the default is

specified as `<xsl:param name="p"><foo/></xsl:param>`, then it is not specified whether a distinct `foo` element node will be created on each invocation of the template, or whether the same `foo` element node will be used for each invocation. However, it is permissible for the default value to depend on the values of other parameters, or on the evaluation context, in which case the default must effectively be evaluated on each invocation.

The `xsl:param` element has an optional `as` attribute, which specifies the required type of the parameter. The value of the `as` attribute is a SequenceType [XP], as defined in [XPath 2.0].

If the `as` attribute is specified, then the supplied value of the parameter is converted to the required type, using the function conversion rules.

[ERR XTTE0590] It is a type error if the conversion of the supplied value of a parameter to its required type fails.

If the `as` attribute is omitted, the supplied value of the parameter is used directly, and no conversion takes place.

The optional `required` attribute may be used to indicate that a parameter is mandatory. This attribute may be specified for stylesheet parameters and for template parameters; it MUST NOT be specified for function parameters, which are always mandatory. A parameter is mandatory if it is a function parameter or if the `required` attribute is present and has the value `yes`. Otherwise, the parameter is optional. If the parameter is mandatory, then the `xsl:param` element MUST be empty and MUST NOT have a `select` attribute.

[ERR XTTE0600] If a default value is given explicitly, that is, if there is either a `select` attribute or a non-empty sequence constructor, then it is a type error if the default value cannot be converted to the required type, using the function conversion rules.

If an optional parameter has no `select` attribute and has an empty sequence constructor, and if there is no `as` attribute, then the default value of the parameter is a zero length string.

[ERR XTDE0610] If an optional parameter has no `select` attribute and has an empty sequence constructor, and if there is an `as` attribute, then the default value of the parameter is an empty sequence. If the empty sequence is not a valid instance of the required type defined in the `as` attribute, then the parameter is treated as a required parameter, which means that it is a non-recoverable dynamic error if the caller supplies no value for the parameter.

Note:

The effect of these rules is that specifying `<xsl:param name="p" as="xs:date" select="2"/>` is an error, but if the default value of the parameter is never used, then the processor has discretion whether or not to report the error. By contrast, `<xsl:param name="p" as="xs:date"/>` is treated as if `required="yes"` had been specified: the empty sequence is not a valid instance of `xs:date`, so in effect there is no default value and the parameter is therefore treated as being mandatory.

The optional `tunnel` attribute may be used to indicate that a parameter is a tunnel parameter. The default is `no`; the value `yes` may be specified only for template parameters. Tunnel parameters are described in *10.1.2 Tunnel Parameters*

9.3 Values of Variables and Parameters

A variable-binding element may specify the supplied value of the variable or parameter in four different ways.

- If the variable-binding element has a `select` attribute, then the value of the attribute MUST be an expression and the supplied value of the variable is the value that results from evaluating the expression. In this case, the content of the variable-binding element MUST be empty.
- If the variable-binding element has empty content and has neither a `select` attribute nor an `as` attribute, then the supplied value of the variable is a zero-length string. Thus

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

  is equivalent to

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

- If a variable-binding element has no `select` attribute and has non-empty content (that is, the variable-binding element has one or more child nodes), and has no `as` attribute, then the content of the variable-binding element specifies the supplied value. The content of the variable-binding element is a sequence constructor; a new document is constructed with a document node having as its children the sequence of nodes that results from evaluating the sequence constructor and then applying the rules given in *5.7.1 Constructing Complex Content*. The value of the variable is then a singleton sequence containing this document node. For further information, see *9.4 Creating implicit document nodes*.
- If a variable-binding element has an `as` attribute but no `select` attribute, then the supplied value is the sequence that results from evaluating the (possibly empty) sequence constructor contained within the variable-binding element (see *5.7 Sequence Constructors*).

These combinations are summarized in the table below.

select attribute	as attribute	content	Effect
present	absent	empty	Value is obtained by evaluating the `select` attribute
present	present	empty	Value is obtained by evaluating the `select` attribute, adjusted to the type required by the `as` attribute
present	absent	present	Static error
present	present	present	Static error
absent	absent	empty	Value is a zero-length string
absent	present	empty	Value is an empty sequence, provided the `as` attribute permits an empty sequence
absent	absent	present	Value is a document node whose content is obtained by evaluating the sequence constructor

absent	present	present	Value is obtained by evaluating the sequence constructor, adjusted to the type required by the `as` attribute

[ERR XTSE0620] It is a static error if a variable-binding element has a `select` attribute and has non-empty content.

Example: Values of Variables

The value of the following variable is the sequence of integers (1, 2, 3):

```
<xsl:variable name="i" as="xs:integer*" select="1 to 3"/>
```

The value of the following variable is an integer, assuming that the attribute `@size` exists, and is annotated either as an integer, or as `xs:untypedAtomic`:

```
<xsl:variable name="i" as="xs:integer" select="@size"/>
```

The value of the following variable is a zero-length string:

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

The value of the following variable is document node containing an empty element as a child:

```
<xsl:variable name="doc"><c/></xsl:variable>
```

The value of the following variable is sequence of integers (2, 4, 6):

```
<xsl:variable name="seq" as="xs:integer*">
  <xsl:for-each select="1 to 3">
    <xsl:sequence select=".*2"/>
  </xsl:for-each>
</xsl:variable>
```

The value of the following variable is sequence of parentless attribute nodes:

```
<xsl:variable name="attset" as="attribute()+">
  <xsl:attribute name="x">2</xsl:attribute>
  <xsl:attribute name="y">3</xsl:attribute>
  <xsl:attribute name="z">4</xsl:attribute>
</xsl:variable>
```

The value of the following variable is an empty sequence:

```
<xsl:variable name="empty" as="empty-sequence()"/>
```

The actual value of the variable depends on the supplied value, as described above, and the required type, which is determined by the value of the `as` attribute.

Example: Pitfalls with Numeric Predicates

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="td[$n]"/>
```

This will output the values of all the `td` elements, space-separated (or in backwards compatibility mode, the value of the first `td` element), because the variable `n` will be bound to a node, not a number. Instead, do one of the following:

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

or

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

or

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

9.4 Creating implicit document nodes

A document node is created implicitly when evaluating an `xsl:variable`, `xsl:param`, or `xsl:with-param` element that has non-empty content and that has no `as` attribute. This element is referred to as the variable-binding element. The value of the variable is a single node, the document node of the temporary tree. The content of the document node is formed from the result of evaluating the sequence constructor contained within the variable-binding element, as described in *5.7.1 Constructing Complex Content*.

Note:

The construct:

```
<xsl:variable name="tree">
  <a/>
</xsl:variable>
```

can be regarded as a shorthand for:

```
<xsl:variable name="tree" as="document-node()">
  <xsl:document validation="preserve">
    <a/>
  </xsl:document>
</xsl:variable>
```

The base URI of the document node is taken from the base URI of the variable binding element in the stylesheet. (See Section 5.2 base-uri Accessor[DM] in [Data Model])

No document-level validation takes place (which means, for example, that there is no checking that ID values are unique). However, type annotations on nodes within the new tree are copied unchanged.

Note:

The base URI of other nodes in the tree is determined by the rules for constructing complex content. The effect of these rules is that the base URI of a node in the temporary tree is determined as if all the nodes in the temporary tree came from a single entity whose URI was the base URI of the variable-binding element. Thus, the base URI of the document node will be equal to the base URI of the variable-binding element; an `xml:base` attribute within the temporary tree will change the base URI for its parent element and that element's descendants, just as it would within a document constructed by parsing.

The `document-uri` and `unparsed-entities` properties of the new document node are set to empty.

A temporary tree is available for processing in exactly the same way as any source document. For example, its nodes are accessible using path expressions, and they can be processed using instructions such as `xsl:apply-templates` and `xsl:for-each`. Also, the `key` and `id` [FO] functions can be used to find nodes within a temporary tree rooted at a document node, provided that at the time the function is called, the context item is a node within the temporary tree.

Example: Two-Phase Transformation

For example, the following stylesheet uses a temporary tree as the intermediate result of a two-phase transformation, using different modes for the two phases (see *6.5 Modes*). Typically, the template rules in module `phase1.xsl` will be declared with `mode="phase1"`, while those in module `phase2.xsl` will be declared with `mode="phase2"`:

```
<xsl:stylesheet
  version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:import href="phase1.xsl"/>
<xsl:import href="phase2.xsl"/>

<xsl:variable name="intermediate">
  <xsl:apply-templates select="/" mode="phase1"/>
</xsl:variable>

<xsl:template match="/">
  <xsl:apply-templates select="$intermediate" mode="phase2"/>
</xsl:template>

</xsl:stylesheet>
```

Note:

The algorithm for matching nodes against template rules is exactly the same regardless which tree the nodes come from. If different template rules are to be used when processing different trees, then unless nodes from different trees can be distinguished by means of patterns, it is a good idea to use modes to ensure that each tree is processed using the appropriate set of template rules.

9.5 Global Variables and Parameters

Both `xsl:variable` and `xsl:param` are allowed as declaration elements: that is, they may appear as children of the `xsl:stylesheet` element.

[DEFINITION: A top-level variable-binding element declares a **global variable** that is visible everywhere (except where it is shadowed by another binding).]

[DEFINITION: A top-level `xsl:param` element declares a **stylesheet parameter**. A stylesheet parameter is a global variable with the additional property that its value can be supplied by the caller when a transformation is initiated.] As described in *9.2 Parameters*, a stylesheet parameter may be declared as being mandatory, or may have a default value specified for use when no value is supplied by the caller. The

mechanism by which the caller supplies a value for a stylesheet parameter is implementation-defined. An XSLT processor MUST provide such a mechanism.

It is an error if no value is supplied for a mandatory stylesheet parameter [see ERR XTDE0050].

If a stylesheet contains more than one binding for a global variable of a particular name, then the binding with the highest import precedence is used.

[ERR XTSE0630] It is a static error if a stylesheet contains more than one binding of a global variable with the same name and same import precedence, unless it also contains another binding with the same name and higher import precedence.

For a global variable or the default value of a stylesheet parameter, the expression or sequence constructor specifying the variable value is evaluated with a singleton focus based on the root node of the tree containing the initial context node. An XPath error will be reported if the evaluation of a global variable or parameter references the context item, context position, or context size when no initial context node is supplied. The values of other components of the dynamic context are the initial values as defined in *5.4.3 Initializing the Dynamic Context* and *5.4.4 Additional Dynamic Context Components used by XSLT*.

Example: A Stylesheet Parameter

The following example declares a global parameter `para-font-size`, which is referenced in an attribute value template.

```
<xsl:param name="para-font-size" as="xs:string">12pt</xsl:param>

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

The implementation must provide a mechanism allowing the user to supply a value for the parameter `para-font-size` when invoking the stylesheet; the value `12pt` acts as a default.

9.6 Local Variables and Parameters

[DEFINITION: As well as being allowed as declaration elements, the `xsl:variable` element is also allowed in sequence constructors. Such a variable is known as a **local variable**.]

[DEFINITION: An `xsl:param` element may appear as a child of an `xsl:template` element, before any non-`xsl:param` children of that element. Such a parameter is known as a **template parameter**. A template parameter is a local variable with the additional property that its value can be set when the template is called, using any of the instructions `xsl:call-template`, `xsl:apply-templates`, `xsl:apply-imports`, or `xsl:next-match`.]

[DEFINITION: An `xsl:param` element may appear as a child of an `xsl:function` element, before any non-`xsl:param` children of that element. Such a parameter is known as a **function parameter**. A function parameter is a local variable with the additional property that its value can be set when the function is called, using a function call in an XPath expression.]

The result of evaluating a local `xsl:variable` or `xsl:param` element (that is, the contribution it makes to the result of the sequence constructor it is part of) is an empty sequence.

9.7 Scope of Variables

For any variable-binding element, there is a region (more specifically, a set of element nodes) of the stylesheet within which the binding is visible. The set of variable bindings in scope for an XPath expression consists of those bindings that are visible at the point in the stylesheet where the expression occurs.

A global variable binding element is visible everywhere in the stylesheet (including other stylesheet modules) except within the `xsl:variable` or `xsl:param` element itself and any region where it is shadowed by another variable binding.

A local variable binding element is visible for all following siblings and their descendants, with two exceptions: it is not visible in any region where it is shadowed by another variable binding, and it is not visible within the subtree rooted at an `xsl:fallback` instruction that is a sibling of the variable binding element. The binding is not visible for the `xsl:variable` or `xsl:param` element itself.

[DEFINITION: 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 not an error if a binding established by a local `xsl:variable` or `xsl:param` shadows a global binding. In this case, the global binding will not be visible in the region of the stylesheet where it is shadowed by the other binding.

Example: Local Variable Shadowing a Global Variable

The following is allowed:

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

It is also not an error if a binding established by a local `xsl:variable` element shadows a binding established by another local `xsl:variable` or `xsl:param`.

Example: Misuse of Variable Shadowing

The following is not an error, but the effect is probably not what was intended. The template outputs `<x value="1"/>`, because the declaration of the inner variable named `$x` has no effect on the value of the outer variable named `$x`.

```
<xsl:variable name="x" select="1"/>
<xsl:template name="foo">
  <xsl:for-each select="1 to 5">
    <xsl:variable name="x" select="$x+1"/>
  </xsl:for-each>
  <x value="{$x}"/>
</xsl:template>
```

Note:

Once a variable has been given a value, the value cannot subsequently be changed. XSLT does not provide an equivalent to the assignment operator available in many procedural programming languages.

This is because an assignment operator 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.

As well as global variables and local variables, an XPath expression may also declare range variables for use locally within an expression. For details, see [XPath 2.0].

Where a reference to a variable occurs in an XPath expression, it is resolved first by reference to range variables that are in scope, then by reference to local variables and parameters, and finally by reference to global variables and parameters. A range variable may shadow a local variable or a global variable. XPath also allows a range variable to shadow another range variable.

9.8 Circular Definitions

[DEFINITION: A **circularity** is said to exist if a construct such as a global variable, an attribute set, or a key is defined in terms of itself. For example, if the expression or sequence constructor 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*. A circularity exists if it is impossible to do this for all global variable definitions.]

Example: Circular Variable Definitions

The following two declarations create a circularity:

```
<xsl:variable name="x" select="$y+1"/>
<xsl:variable name="y" select="$x+1"/>
```

Example: Circularity involving Variables and Functions

The definition of a global variable can be circular even if no other variable is involved. For example the following two declarations (see *10.3 Stylesheet Functions* for an explanation of the `xsl:function` element) also create a circularity:

```
<xsl:variable name="x" select="my:f()"/>

<xsl:function name="my:f">
  <xsl:sequence select="$x"/>
</xsl:function>
```

Example: Circularity involving Variables and Templates

The definition of a variable is also circular if the evaluation of the variable invokes an `xsl:apply-templates` instruction and the variable is referenced in the pattern used in the `match` attribute of any template rule in the stylesheet. For example the following definition is circular:

```
<xsl:variable name="x">
  <xsl:apply-templates select="//param[1]"/>
</xsl:variable>

<xsl:template match="param[$x]">1</xsl:template>
```

Example: Circularity involving Variables and Keys

Similarly, a variable definition is circular if it causes a call on the `key` function, and the definition of that key refers to that variable in its `match` or `use` attributes. So the following definition is circular:

```
<xsl:variable name="x" select="my:f(10)"/>

<xsl:function name="my:f">
  <xsl:param name="arg1"/>
  <xsl:sequence select="key('k', $arg1)"/>
</xsl:function>

<xsl:key name="k" match="item[@code=$x]" use="@desc"/>
```

[ERR XTDE0640] In general, a circularity in a stylesheet is a non-recoverable dynamic error. However, as with all other dynamic errors, an implementation will signal the error only if it actually executes the instructions and expressions that participate in the circularity. Because different implementations may optimize the execution of a stylesheet in different ways, it is implementation-dependent whether a particular circularity will actually be signaled.

For example, in the following declarations, the function declares a local variable `$b`, but it returns a result that does not require the variable to be evaluated. It is implementation-dependent whether the value is actually evaluated, and it is therefore implementation-dependent whether the circularity is signaled as an error:

```
<xsl:variable name="x" select="my:f(1)/>

<xsl:function name="my:f">
  <xsl:param name="a"/>
  <xsl:variable name="b" select="$x"/>
  <xsl:sequence select="$a + 2"/>
</xsl:function>
```

Circularities usually involve global variables or parameters, but they can also exist between key definitions (see *16.3 Keys*), between named attribute sets (see *10.2 Named Attribute Sets*), or between any combination of these constructs. For example, a circularity exists if a key definition invokes a function that references an attribute set that calls the `key` function, supplying the name of the original key definition as an argument.

Circularity is not the same as recursion. Stylesheet functions (see *10.3 Stylesheet Functions*) and named templates (see *10.1 Named Templates*) may call other functions and named templates without restriction. With careless coding, recursion may be non-terminating. Implementations are REQUIRED to signal circularity as a dynamic error, but they are not REQUIRED to detect non-terminating recursion.

10 Callable Components

This section describes three constructs that can be used to provide subroutine-like functionality that can be invoked from anywhere in the stylesheet: named templates (see *10.1 Named Templates*), named attribute sets (see *10.2 Named Attribute Sets*) and stylesheet functions (see *10.3 Stylesheet Functions*).

10.1 Named Templates

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

[DEFINITION: Templates can be invoked by name. An `xsl:template` element with a `name` attribute defines a **named template**.] The value of the `name` attribute is a QName, which is expanded as described in *5.1 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` instruction invokes a template by name; it has a REQUIRED `name` attribute that identifies the template to be invoked. Unlike `xsl:apply-templates`, the `xsl:call-template` instruction does not change the focus.

The `match`, `mode` and `priority` attributes on an `xsl:template` element have no effect when the template is invoked by an `xsl:call-template` instruction. Similarly, the `name` attribute on an `xsl:template` element has no effect when the template is invoked by an `xsl:apply-templates` instruction.

[ERR XTSE0650] It is a static error if a stylesheet contains an `xsl:call-template` instruction whose `name` attribute does not match the `name` attribute of any `xsl:template` in the stylesheet.

[ERR XTSE0660] It is a static error if a stylesheet contains more than one template with the same name and the same import precedence, unless it also contains a template with the same name and higher import precedence.

The target template for an `xsl:call-template` instruction is the template whose `name` attribute matches the `name` attribute of the `xsl:call-template` instruction and that has higher import precedence than any other template with this name. The result of evaluating an `xsl:call-template` instruction is the sequence produced by evaluating the sequence constructor contained in its target template (see *5.7 Sequence Constructors*).

10.1.1 Passing Parameters to Templates

```
<xsl:with-param
  name = qname
  select? = expression
  as? = sequence-type
  tunnel? = "yes" | "no">
  <!-- Content: sequence-constructor -->
</xsl:with-param>
```

Parameters are passed to templates using the `xsl:with-param` element. The REQUIRED `name` attribute specifies the name of the template 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 *5.1 Qualified Names*.

`xsl:with-param` is allowed within `xsl:call-template`, `xsl:apply-templates`, `xsl:apply-imports`, and `xsl:next-match`.

[ERR XTSE0670] It is a static error if a single xsl:call-template, xsl:apply-templates, xsl:apply-imports, or xsl:next-match element contains two or more xsl:with-param elements with matching name attributes.

The value of the parameter is specified in the same way as for xsl:variable and xsl:param (see *9.3 Values of Variables and Parameters*), taking account of the values of the select and as attributes and the content of the xsl:with-param element, if any.

Note:

It is possible to have an as attribute on the xsl:with-param element that differs from the as attribute on the corresponding xsl:param element describing the formal parameters of the called template.

In this situation, the supplied value of the parameter will first be processed according to the rules of the as attribute on the xsl:with-param element, and the resulting value will then be further processed according to the rules of the as attribute on the xsl:param element.

For example, suppose the supplied value is a node with type annotation xs:untypedAtomic, and the xsl:with-param element specifies as="xs:integer", while the xsl:param element specifies as="xs:double". Then the node will first be atomized and the resulting untyped atomic value will be cast to xs:integer. If this succeeds, the xs:integer will then be promoted to an xs:double.

The focus used for computing the value specified by the xsl:with-param element is the same as that used for the xsl:apply-templates, xsl:apply-imports, xsl:next-match, or xsl:call-template element within which it occurs.

[ERR XTSE0680] In the case of xsl:call-template, it is a static error to pass a non-tunnel parameter named *x* to a template that does not have a template parameter named *x*, unless backwards compatible behavior is enabled for the xsl:call-template instruction. This is not an error in the case of xsl:apply-templates, xsl:apply-imports, and xsl:next-match; in these cases the parameter is simply ignored.

The optional tunnel attribute may be used to indicate that a parameter is a tunnel parameter. The default is no. Tunnel parameters are described in *10.1.2 Tunnel Parameters*

[ERR XTSE0690] It is a static error if a template that is invoked using xsl:call-template declares a template parameter specifying required="yes" and not specifying tunnel="yes", if no value for this parameter is supplied by the calling instruction.

[ERR XTDE0700] In other cases, it is a non-recoverable dynamic error if the template that is invoked declares a template parameter with required="yes" and no value for this parameter is supplied by the calling instruction.

Example: A Named Template

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>
```

Note:

Arguments to stylesheet functions are supplied as part of an XPath function call: see *10.3 Stylesheet Functions*

10.1.2 Tunnel Parameters

[DEFINITION: A parameter passed to a template may be defined as a **tunnel parameter**. Tunnel parameters have the property that they are automatically passed on by the called template to any further templates that it calls, and so on recursively.] Tunnel parameters thus allow values to be set that are accessible during an entire phase of stylesheet processing, without the need for each template that is used during that phase to be aware of the parameter.

Note:

Tunnel parameters are conceptually similar to dynamically-scoped variables in some functional programming languages.

A tunnel parameter is created by using an xsl:with-param element that specifies tunnel="yes". A template that requires access to the value of a tunnel parameter must declare it using an xsl:param element that also specifies tunnel="yes".

On any template call using an xsl:apply-templates, xsl:call-template, xsl:apply-imports or xsl:next-match instruction, a set of tunnel parameters is passed from the calling template to the called template. This set consists of any parameters explicitly created using <xsl:with-param tunnel="yes">, overlaid on a base set of tunnel parameters. If the xsl:apply-templates, xsl:call-template, xsl:apply-imports or xsl:next-match instruction has an xsl:template declaration as an ancestor element in the stylesheet, then the base set consists of the tunnel parameters that were passed to that template; otherwise (for example, if the instruction is within a global variable declaration, an attribute set declaration, or a stylesheet function), the base set is empty. If a parameter created using <xsl:with-param tunnel="yes"> has the same expanded-QName as a parameter in the base set, then the parameter created using xsl:with-param overrides the parameter in the base set; otherwise, the parameter created using xsl:with-param is added to the base set.

When a template accesses the value of a tunnel parameter by declaring it with xsl:param tunnel="yes", this does not remove the parameter from the base set of tunnel parameters that is passed on to any templates called by this template.

Two sibling xsl:with-param elements must have distinct parameter names, even if one is a tunnel parameter and the other is not. Equally,

two sibling xsl:param elements representing template parameters must have distinct parameter names, even if one is a tunnel parameter and the other is not. However, the tunnel parameters that are implicitly passed in a template call may have names that duplicate the names of non-tunnel parameters that are explicitly passed on the same call.

Tunnel parameters are not passed in calls to stylesheet functions.

All other options of xsl:with-param and xsl:param are available with tunnel parameters just as with non-tunnel parameters. For example, parameters may be declared as mandatory or optional, a default value may be specified, and a required type may be specified. If any conversion is required from the supplied value of a tunnel parameter to the required type specified in xsl:param, then the converted value is used within the receiving template, but the value that is passed on in any further template calls is the original supplied value before conversion. Equally, any default value is local to the template: specifying a default value for a tunnel parameter does not change the set of tunnel parameters that is passed on in further template calls.

The set of tunnel parameters that is passed to the initial template is empty.

Tunnel parameters are passed unchanged through a built-in template rule (see *6.6 Built-in Template Rules*).

Example: Using Tunnel Parameters

Suppose that the equations in a scientific paper are to be sequentially numbered, but that the format of the number depends on the context in which the equations appear. It is possible to reflect this using a rule of the form:

```
<xsl:template match="equation">
  <xsl:param name="equation-format" select="'(1)'" tunnel="yes"/>
  <xsl:number level="any" format="{$equation-format}"/>
</xsl:template>
```

At any level of processing above this level, it is possible to determine how the equations will be numbered, for example:

```
<xsl:template match="appendix">
  ...
  <xsl:apply-templates>
    <xsl:with-param name="equation-format" select="'[i]'" tunnel="yes"/>
  </xsl:apply-templates>
  ...
</xsl:template>
```

The parameter value is passed transparently through all the intermediate layers of template rules until it reaches the rule with match="equation". The effect is similar to using a global variable, except that the parameter can take different values during different phases of the transformation.

10.2 Named Attribute Sets

```
<!-- Category: declaration -->
<xsl:attribute-set
  name = qname
  use-attribute-sets? = qnames>
  <!-- Content: xsl:attribute* -->
</xsl:attribute-set>
```

[DEFINITION: The xsl:attribute-set element defines a named **attribute set**: that is, a collection of attribute definitions that can be used repeatedly on different constructed elements.]

The REQUIRED name attribute specifies the name of the attribute set. The value of the name attribute is a QName, which is expanded as described in *5.1 Qualified Names*. The content of the xsl:attribute-set element consists of zero or more xsl:attribute instructions that are evaluated to produce the attributes in the set.

The result of evaluating an attribute set is a sequence of attribute nodes. Evaluating the same attribute set more than once can produce different results, because although an attribute set does not have parameters, it may contain expressions or instructions whose value depends on the evaluation context.

Attribute sets are used by specifying a use-attribute-sets attribute on the xsl:element or xsl:copy instruction, or by specifying an xsl:use-attribute-sets attribute on a literal result element. An attribute set may be defined in terms of other attribute sets by using the use-attribute-sets attribute on the xsl:attribute-set element itself. The value of the [xsl:]use-attribute-sets attribute is in each case a whitespace-separated list of names of attribute sets. Each name is specified as a QName, which is expanded as described in *5.1 Qualified Names*.

Specifying a use-attribute-sets attribute is broadly equivalent to adding xsl:attribute instructions for each of the attributes in each of the named attribute sets to the beginning of the content of the instruction with the [xsl:]use-attribute-sets attribute, in the same order in which the names of the attribute sets are specified in the use-attribute-sets attribute.

More formally, an xsl:use-attribute-sets attribute is expanded using the following recursive algorithm, or any algorithm that produces the same results:

- The value of the attribute is tokenized as a list of QNames.
- Each QName in the list is processed, in order, as follows:
  - The QName must match the name attribute of one or more xsl:attribute-set declarations in the stylesheet.
  - Each xsl:attribute-set declaration whose name matches is processed as follows. Where two such declarations have different import precedence, the one with lower import precedence is processed first. Where two declarations have the same import precedence, they are processed in declaration order.
    - If the xsl:attribute-set declaration has a use-attribute-sets attribute, the attribute is expanded by applying this

algorithm recursively.

- If the `xsl:attribute-set` declaration contains one or more `xsl:attribute` instructions, these instructions are evaluated (following the rules for evaluating a sequence constructor: see *5.7 Sequence Constructors*) to produce a sequence of attribute nodes. These attribute nodes are appended to the result sequence.

The `xsl:attribute` instructions are evaluated using the same focus as is used for evaluating the element that is the parent of the `[xsl:]use-attribute-sets` attribute forming the initial input to the algorithm. However, the static context for the evaluation depends on the position of the `xsl:attribute` instruction in the stylesheet: thus, only local variables declared within an `xsl:attribute` instruction, and global variables, are visible.

The set of attribute nodes produced by expanding `xsl:use-attribute-sets` may include several attributes with the same name. When the attributes are added to an element node, only the last of the duplicates will take effect.

The way in which each instruction uses the results of expanding the `[xsl:]use-attribute-sets` attribute is described in the specification for the relevant instruction: see *11.1 Literal Result Elements*, *11.2 Creating Element Nodes Using xsl:element* , and *11.9 Copying Nodes*.

[ERR XTSE0710] It is a static error if the value of the `use-attribute-sets` attribute of an `xsl:copy`, `xsl:element`, or `xsl:attribute-set` element, or the `xsl:use-attribute-sets` attribute of a literal result element, is not a whitespace-separated sequence of QNames, or if it contains a QName that does not match the `name` attribute of any `xsl:attribute-set` declaration in the stylesheet.

[ERR XTSE0720] It is a static error if an `xsl:attribute-set` element directly or indirectly references itself via the names contained in the `use-attribute-sets` attribute.

Each attribute node produced by expanding an attribute set has a type annotation determined by the rules for the `xsl:attribute` instruction that created the attribute node: see *11.3.1 Setting the Type Annotation for a Constructed Attribute Node*. These type annotations may be preserved, stripped, or replaced as determined by the rules for the instruction that creates the element in which the attributes are used.

Attribute sets are used as follows:

- The `xsl:copy` and `xsl:element` instructions have an `use-attribute-sets` attribute. The sequence of attribute nodes produced by evaluating this attribute is prepended to the sequence produced by evaluating the sequence constructor contained within the instruction.
- Literal result elements allow an `xsl:use-attribute-sets` attribute, which is evaluated in the same way as the `use-attribute-sets` attribute of `xsl:element` and `xsl:copy`. The sequence of attribute nodes produced by evaluating this attribute is prepended to the sequence of attribute nodes produced by evaluating the attributes of the literal result element, which in turn is prepended to the sequence produced by evaluating the sequence constructor contained with the literal result element.

Example: Using Attribute Sets

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

```
<xsl:template match="chapter/heading">
  <fo:block font-stretch="condensed" 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>
```

Example: Overriding Attributes in an Attribute Set

The following example creates a named attribute set `base-style` and uses it in a template rule with multiple specifications of the attributes:

font-family

is specified only in the attribute set

font-size

is specified in the attribute set, is specified on the literal result element, and in an `xsl:attribute` instruction

font-style

is specified in the attribute set, and on the literal result element

font-weight

is specified in the attribute set, and in an `xsl:attribute` instruction

Stylesheet fragment:

```
<xsl:attribute-set name="base-style">
  <xsl:attribute name="font-family">Univers</xsl:attribute>
  <xsl:attribute name="font-size">10pt</xsl:attribute>
  <xsl:attribute name="font-style">normal</xsl:attribute>
  <xsl:attribute name="font-weight">normal</xsl:attribute>
```

```
  </xsl:attribute-set>

  <xsl:template match="o">
    <fo:block xsl:use-attribute-sets="base-style"
              font-size="12pt"
              font-style="italic">
      <xsl:attribute name="font-size">14pt</xsl:attribute>
      <xsl:attribute name="font-weight">bold</xsl:attribute>
      <xsl:apply-templates/>
    </fo:block>
  </xsl:template>
```

Result:

```
  <fo:block font-family="Univers"
            font-size="14pt"
            font-style="italic"
            font-weight="bold">
  ...
  </fo:block>
```

10.3 Stylesheet Functions

[DEFINITION: An `xsl:function` declaration declares the name, parameters, and implementation of a **stylesheet function** that can be called from any XPath expression within the stylesheet.]

```
<!-- Category: declaration -->
<xsl:function
  name = qname
  as? = sequence-type
  override? = "yes" | "no">
  <!-- Content: (xsl:param*, sequence-constructor) -->
</xsl:function>
```

The `xsl:function` declaration defines a stylesheet function that can be called from any XPath expression used in the stylesheet (including an XPath expression used within a predicate in a pattern). The `name` attribute specifies the name of the function. The value of the `name` attribute is a QName, which is expanded as described in *5.1 Qualified Names*.

An `xsl:function` declaration can only appear as a top-level element in a stylesheet module.

[ERR XTSE0740] A stylesheet function MUST have a prefixed name, to remove any risk of a clash with a function in the default function namespace. It is a static error if the name has no prefix..

> **Note:**
>
> To prevent the namespace declaration used for the function name appearing in the result document, use the `exclude-result-prefixes` attribute on the `xsl:stylesheet` element: see *11.1.3 Namespace Nodes for Literal Result Elements*.
>
> The prefix MUST NOT refer to a reserved namespace: [see ERR XTSE0080]

The content of the `xsl:function` element consists of zero or more `xsl:param` elements that specify the formal arguments of the function, followed by a sequence constructor that defines the value to be returned by the function.

[DEFINITION: The **arity** of a stylesheet function is the number of `xsl:param` elements in the function definition.] Optional arguments are not allowed.

[ERR XTSE0760] Because arguments to a stylesheet function call MUST all be specified, the `xsl:param` elements within an `xsl:function` element MUST NOT specify a default value: this means they MUST be empty, and MUST NOT have a `select` attribute.

A stylesheet function is included in the *in-scope functions* of the static context for all XPath expressions used in the stylesheet, unless

- there is another stylesheet function with the same name and arity, and higher import precedence, or
- the `override` attribute has the value `no` and there is already a function with the same name and arity in the in-scope functions.

The optional `override` attribute defines what happens if this function has the same name and arity as a function provided by the implementer or made available in the static context using an implementation-defined mechanism. If the `override` attribute has the value `yes`, then this function is used in preference; if it has the value `no`, then the other function is used in preference. The default value is `yes`.

> **Note:**
>
> Specifying `override="yes"` ensures interoperable behavior: the same code will execute with all processors. Specifying `override="no"` is useful when writing a fallback implementation of a function that is available with some processors but not others: it allows the vendor's implementation of the function (or a user's implementation written as an extension function) to be used in preference to the stylesheet implementation, which is useful when the extension function is more efficient.
>
> The `override` attribute does *not* affect the rules for deciding which of several stylesheet functions with the same name and arity takes precedence.

[ERR XTSE0770] It is a static error for a stylesheet to contain two or more functions with the same expanded-QName, the same arity, and the same import precedence, unless there is another function with the same expanded-QName and arity, and a higher import precedence.

As defined in XPath, the function that is executed as the result of a function call is identified by looking in the in-scope functions of the static context for a function whose name and arity matches the name and number of arguments in the function call.

> **Note:**

Functions are not polymorphic. Although the XPath function call mechanism allows two functions to have the same name and different arity, it does not allow them to be distinguished by the types of their arguments.

The optional `as` attribute indicates the required type of the result of the function. The value of the `as` attribute is a SequenceType [XP], as defined in [XPath 2.0].

[ERR XTTE0780] If the `as` attribute is specified, then the result evaluated by the sequence constructor (see *5.7 Sequence Constructors*) is converted to the required type, using the function conversion rules. It is a type error if this conversion fails. If the `as` attribute is omitted, the calculated result is used as supplied, and no conversion takes place.

If a stylesheet function has been defined with a particular expanded-QName, then a call on `function-available` will return true when called with an argument that is a lexical QName that expands to this same expanded-QName.

The `xsl:param` elements define the formal arguments to the function. These are interpreted positionally. When the function is called using a function-call in an XPath expression, the first argument supplied is assigned to the first `xsl:param` element, the second argument supplied is assigned to the second `xsl:param` element, and so on.

The `as` attribute of the `xsl:param` element defines the required type of the parameter. The rules for converting the values of the actual arguments supplied in the function call to the types required by each `xsl:param` element are defined in [XPath 2.0]. The rules that apply are those for the case where XPath 1.0 compatibility mode is set to `false`.

[ERR XTTE0790] If the value of a parameter to a stylesheet function cannot be converted to the required type, a type error is signaled.

If the `as` attribute is omitted, no conversion takes place and any value is accepted.

Within the body of a stylesheet function, the focus is initially undefined; this means that any attempt to reference the context item, context position, or context size is a non-recoverable dynamic error. [XPDY0002]

It is not possible within the body of the stylesheet function to access the values of local variables that were in scope in the place where the function call was written. Global variables, however, remain available.

Example: A Stylesheet Function

The following example creates a recursive stylesheet function named `str:reverse` that reverses the words in a supplied sentence, and then invokes this function from within a template rule.

```
<xsl:transform
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:str="http://example.com/namespace"
  version="2.0"
  exclude-result-prefixes="str">

<xsl:function name="str:reverse" as="xs:string">
  <xsl:param name="sentence" as="xs:string"/>
  <xsl:sequence
     select="if (contains($sentence, ' '))
             then concat(str:reverse(substring-after($sentence, ' ')),
                         ' ',
                         substring-before($sentence, ' '))
             else $sentence"/>
</xsl:function>

<xsl:template match="/">
<output>
  <xsl:value-of select="str:reverse('DOG BITES MAN')"/>
</output>
</xsl:template>

</xsl:transform>
```

An alternative way of writing the same function is to implement the conditional logic at the XSLT level, thus:

```
<xsl:function name="str:reverse" as="xs:string">
  <xsl:param name="sentence" as="xs:string"/>
  <xsl:choose>
    <xsl:when test="contains($sentence, ' ')">
      <xsl:sequence select="concat(str:reverse(substring-after($sentence, ' ')),
                                   ' ',
                                   substring-before($sentence, ' '))"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:sequence select="$sentence"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:function>
```

Example: Declaring the Return Type of a Function

The following example illustrates the use of the `as` attribute in a function definition. It returns a string containing the representation of its integer argument, expressed as a roman numeral. For example, the function call `num:roman(7)` will return the string `"vii"`. This example uses the `xsl:number` instruction, described in *12 Numbering*. The `xsl:number` instruction returns a text node, and the function conversion rules are invoked to convert this text node to the type declared in the `xsl:function` element, namely `xs:string`. So the text node is atomized to a string.

```
<xsl:function name="num:roman" as="xs:string">
  <xsl:param name="value" as="xs:integer"/>
  <xsl:number value="$value" format="i"/>
</xsl:function>
```

11 Creating Nodes and Sequences

This section describes instructions that directly create new nodes, or sequences of nodes and atomic values.

11.1 Literal Result Elements

[DEFINITION: In a sequence constructor, an element in the stylesheet that does not belong to the XSLT namespace and that is not an extension instruction (see *18.2 Extension Instructions*) is classified as a **literal result element**.] A literal result element is evaluated to construct a new element node with the same expanded-QName (that is, the same namespace URI, local name, and namespace prefix). The result of evaluating a literal result element is a node sequence containing one element, the newly constructed element node.

The content of the element is a sequence constructor (see *5.7 Sequence Constructors*). The sequence obtained by evaluating this sequence constructor, after prepending any attribute nodes produced as described in *11.1.2 Attribute Nodes for Literal Result Elements* and namespace nodes produced as described in *11.1.3 Namespace Nodes for Literal Result Elements*, is used to construct the content of the element, following the rules in *5.7.1 Constructing Complex Content*

The base URI of the new element is copied from the base URI of the literal result element in the stylesheet, unless the content of the new element includes an `xml:base` attribute, in which case the base URI of the new element is the value of that attribute, resolved (if it is a relative URI) against the base URI of the literal result element in the stylesheet. (Note, however, that this is only relevant when creating a parentless element. When the literal result element is copied to form a child of an element or document node, the base URI of the new copy is taken from that of its new parent.)

11.1.1 Setting the Type Annotation for Literal Result Elements

The attributes `xsl:type` and `xsl:validation` may be used on a literal result element to invoke validation of the contents of the element against a type definition or element declaration in a schema, and to determine the type annotation that the new element node will carry. These attributes also affect the type annotation carried by any elements and attributes that have the new element node as an ancestor. These two attributes are both optional, and if one is specified then the other MUST be omitted.

The value of the `xsl:validation` attribute, if present, must be one of the values `strict`, `lax`, `preserve`, or `strip`. The value of the `xsl:type` attribute, if present, must be a QName identifying a type definition that is present in the in-scope schema components for the stylesheet. Neither attribute may be specified as an attribute value template. The effect of these attributes is described in *19.2 Validation*.

11.1.2 Attribute Nodes for Literal Result Elements

Attribute nodes for a literal result element may be created by including `xsl:attribute` instructions within the sequence constructor. Additionally, attribute nodes are created corresponding to the attributes of the literal result element in the stylesheet, and as a result of expanding the `xsl:use-attribute-sets` attribute of the literal result element, if present.

The sequence that is used to construct the content of the literal result element (as described in *5.7.1 Constructing Complex Content*) is the concatenation of the following four sequences, in order:

1. The sequence of namespace nodes produced as described in *11.1.3 Namespace Nodes for Literal Result Elements*.
2. The sequence of attribute nodes produced by expanding the `xsl:use-attribute-sets` attribute (if present) following the rules given in *10.2 Named Attribute Sets*
3. The attributes produced by processing the attributes of the literal result element itself, other than attributes in the XSLT namespace. The way these are processed is described below.
4. The sequence produced by evaluating the contained sequence constructor, if the element is not empty.

Note:

The significance of this order is that an attribute produced by an `xsl:attribute`, `xsl:copy`, or `xsl:copy-of` instruction in the content of the literal result element takes precedence over an attribute produced by expanding an attribute of the literal result element itself, which in turn takes precedence over an attribute produced by expanding the `xsl:use-attribute-sets` attribute. This is because of the rules in *5.7.1 Constructing Complex Content*, which specify that when two or more attributes in the sequence have the same name, all but the last of the duplicates are discarded.

Although the above rules place namespace nodes before attributes, this is not strictly necessary, because the rules in *5.7.1 Constructing Complex Content* allow the namespaces and attributes to appear in any order so long as both come before other kinds of node. The order of namespace nodes and attribute nodes in the sequence has no effect on the relative position of the nodes in document order once they are added to a tree.

Each attribute of the literal result element, other than an attribute in the XSLT namespace, is processed to produce an attribute for the element in the result tree.

The value of such an attribute is interpreted as an attribute value template: it can therefore contain expressions contained in curly brackets (`{}`). The new attribute node will have the same expanded-QName (that is, the same namespace URI, local name, and namespace prefix) as the attribute in the stylesheet tree, and its string value will be the same as the effective value of the attribute in the stylesheet tree. The type annotation on the attribute will initially be `xs:untypedAtomic`, and the typed value of the attribute node will be the same as its string value.

Note:

The eventual type annotation of the attribute in the result tree depends on the `xsl:validation` and `xsl:type` attributes of the parent literal result element, and on the instructions used to create its ancestor elements. If the `xsl:validation` attribute is set to `preserve` or `strip`, the type annotation will be `xs:untypedAtomic`, and the typed value of the attribute node will be the same as its string value. If the `xsl:validation` attribute is set to `strict` or `lax`, or if the `xsl:type` attribute is used, the type annotation on the attribute will be set as a result of the schema validation process applied to the parent element. If neither attribute is present, the type annotation on the attribute will be `xs:untypedAtomic`.

If the name of a constructed attribute is `xml:id`, the processor must perform attribute value normalization by effectively applying the `normalize-space`[FO] function to the value of the attribute, and the resulting attribute node must be given the `is-id` property.

[ERR XTRE0795] It is a recoverable dynamic error if the name of a constructed attribute is `xml:space` and the value is not either `default` or `preserve`. The optional recovery action is to construct the attribute with the value as requested.. This applies whether the attribute is constructed using a literal result element, or by using the `xsl:attribute`, `xsl:copy`, or `xsl:copy-of` instructions.

Note:

The `xml:base`, `xml:lang`, `xml:space`, and `xml:id` attributes have two effects in XSLT. They behave as standard XSLT attributes, which means for example that if they appear on a literal result element, they will be copied to the result tree in the same way as any other attribute. In addition, they have their standard meaning as defined in the core XML specifications. Thus, an `xml:base` attribute in the stylesheet affects the base URI of the element on which it appears, and an `xml:space` attribute affects the interpretation of whitespace text nodes within that element. One consequence of this is that it is inadvisable to write these attributes as attribute value templates: although an XSLT processor will understand this notation, the XML parser will not. See also *11.1.4 Namespace Aliasing* which describes how to use `xsl:namespace-alias` with these attributes.

The same is true of the schema-defined attributes `xsi:type`, `xsi:nil`, `xsi:noNamespaceSchemaLocation`, and `xsi:schemaLocation`. If the stylesheet is processed by a schema processor, these attributes will be recognized and interpreted by the schema processor, but in addition the XSLT processor treats them like any other attribute on a literal result element: that is, their effective value (after expanding attribute value templates) is copied to the result tree in the same way as any other attribute. If the result tree is validated, the copied attributes will again be recognized and interpreted by the schema processor.

None of these attributes will be generated in the result tree unless the stylesheet writes them to the result tree explicitly, in the same way as any other attribute.

[ERR XTSE0805] It is a static error if an attribute on a literal result element is in the XSLT namespace, unless it is one of the attributes explicitly defined in this specification.

Note:

If there is a need to create attributes in the XSLT namespace, this can be achieved using `xsl:attribute`, or by means of the `xsl:namespace-alias` declaration.

11.1.3 Namespace Nodes for Literal Result Elements

The created element node will 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 designated as an **excluded namespace**. Special considerations apply to aliased namespaces: see *11.1.4 Namespace Aliasing*

The following namespaces are designated as excluded namespaces:

- The XSLT namespace URI (`http://www.w3.org/1999/XSL/Transform`)
- A namespace URI declared as an extension namespace (see *18.2 Extension Instructions*)
- A namespace URI designated by using an `[xsl:]exclude-result-prefixes` attribute either on the literal result element itself or on an ancestor element. The attribute MUST be in the XSLT namespace only if its parent element is *not* in the XSLT namespace.

  The value of the attribute is either `#all`, or a whitespace-separated list of tokens, each of which is either a namespace prefix or `#default`. The namespace bound to each of the prefixes is designated as an excluded namespace.

  [ERR XTSE0808] It is a static error if a namespace prefix is used within the `[xsl:]exclude-result-prefixes` attribute and there is no namespace binding in scope for that prefix.

  The default namespace of the parent element of the `[xsl:]exclude-result-prefixes` attribute (see Section 6.2 Element Nodes[DM]) may be designated as an excluded namespace by including `#default` in the list of namespace prefixes.

  [ERR XTSE0809] It is a static error if the value `#default` is used within the `[xsl:]exclude-result-prefixes` attribute and the parent element of the `[xsl:]exclude-result-prefixes` attribute has no default namespace.

  The value `#all` indicates that all namespaces that are in scope for the stylesheet element that is the parent of the `[xsl:]exclude-result-prefixes` attribute are designated as excluded namespaces.

  The designation of a namespace as an excluded namespace is effective within the subtree of the stylesheet module rooted at the element bearing the `[xsl:]exclude-result-prefixes` attribute; a subtree rooted at an `xsl:stylesheet` element does not include any stylesheet modules imported or included by children of that `xsl:stylesheet` element.

The excluded namespaces, as described above, *only* affect namespace nodes copied from the stylesheet when processing a literal result element. There is no guarantee that an excluded namespace will not appear on the result tree for some other reason. Namespace nodes are also written to the result tree as part of the process of namespace fixup (see *5.7.3 Namespace Fixup*), or as the result of instructions such as `xsl:copy` and `xsl:element`.

Note:

When a stylesheet uses a namespace declaration only for the purposes of addressing a source tree, specifying the prefix in the `[xsl:]exclude-result-prefixes` attribute will avoid superfluous namespace declarations in the serialized result tree. The attribute is also useful to prevent namespaces used solely for the naming of stylesheet functions or extension functions from appearing in the serialized result tree.

Example: Excluding Namespaces from the Result Tree

For example, consider the following stylesheet:

```
<xsl:stylesheet xsl:version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:a="a.uri"
  xmlns:b="b.uri">
  exclude-result-prefixes="#all">
```

```
  <xsl:template match="/">
    <foo xmlns:c="c.uri" xmlns:d="d.uri" xmlns:a2="a.uri"
         xsl:exclude-result-prefixes="c"/>
  </xsl:template>

  </xsl:stylesheet>
```

The result of this stylesheet will be:

```
  <foo xmlns:d="d.uri"/>
```

The namespaces `a.uri` and `b.uri` are excluded by virtue of the `exclude-result-prefixes` attribute on the `xsl:stylesheet` element, and the namespace `c.uri` is excluded by virtue of the `xsl:exclude-result-prefixes` attribute on the `foo` element. The setting `#all` does not affect the namespace `d.uri` because `d.uri` is not an in-scope namespace for the `xsl:stylesheet` element. The element in the result tree does not have a namespace node corresponding to `xmlns:a2="a.uri"` because the effect of `exclude-result-prefixes` is to designate the namespace URI `a.uri` as an excluded namespace, irrespective of how many prefixes are bound to this namespace URI.

If the stylesheet is changed so that the literal result element has an attribute `b:bar="3"`, then the element in the result tree will typically have a namespace declaration `xmlns:b="b.uri"` (the processor may choose a different namespace prefix if this is necessary to avoid conflicts). The `exclude-result-prefixes` attribute makes `b.uri` an excluded namespace, so the namespace node is not automatically copied from the stylesheet, but the presence of an attribute whose name is in the namespace `b.uri` forces the namespace fixup process (see *5.7.3 Namespace Fixup*) to introduce a namespace node for this namespace.

A literal result element may have an optional `xsl:inherit-namespaces` attribute, with the value `yes` or `no`. The default value is `yes`. If the value is set to `yes`, or is omitted, then the namespace nodes created for the newly constructed element are copied to the children and descendants of the newly constructed element, as described in *5.7.1 Constructing Complex Content*. If the value is set to `no`, then these namespace nodes are not automatically copied to the children. This may result in namespace undeclarations (such as `xmlns=""` or, in the case of XML 1.1, `xmlns:p=""`) appearing on the child elements when a final result tree is serialized.

11.1.4 Namespace Aliasing

When a stylesheet is used to define a transformation whose output is itself a stylesheet module, or in certain other cases where the result document uses namespaces that it would be inconvenient to use in the stylesheet, namespace aliasing can be used to declare a mapping between a namespace URI used in the stylesheet and the corresponding namespace URI to be used in the result document.

[DEFINITION: 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**.]

[DEFINITION: The namespace URI that is to be used in the result tree as a substitute for a literal namespace URI is called the **target namespace URI**.]

Either of the literal namespace URI or the target namespace URI can be *null*: this is treated as a reference to the set of names that are in no namespace.

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

[DEFINITION: A stylesheet can use the `xsl:namespace-alias` element to declare that a literal namespace URI is being used as an **alias** for a target namespace URI.]

The effect is that when names in the namespace identified by the literal namespace URI are copied to the result tree, the namespace URI in the result tree will be the target namespace URI, instead of the literal namespace URI. This applies to:

- the namespace URI in the expanded-QName of a literal result element in the stylesheet
- the namespace URI in the expanded-QName of an attribute specified on a literal result element in the stylesheet

Where namespace aliasing changes the namespace URI part of the expanded-QName containing the name of an element or attribute node, the namespace prefix in that expanded-QName is replaced by the prefix indicated by the `result-prefix` attribute of the `xsl:namespace-alias` declaration.

The `xsl:namespace-alias` element declares that the namespace URI bound to the prefix specified by the `stylesheet-prefix` is the literal namespace URI, and the namespace URI bound to the prefix specified by the `result-prefix` attribute is the target namespace URI. 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 no default namespace is in force, specifying `#default` denotes the null namespace URI. This allows elements that are in no namespace in the stylesheet to acquire a namespace in the result document, or vice versa.

If a literal namespace URI is declared to be an alias for multiple different target namespace URIs, then the declaration with the highest import precedence is used.

[ERR XTSE0810] It is a static error if there is more than one such declaration with the same literal namespace URI and the same import precedence and different values for the target namespace URI, unless there is also an `xsl:namespace-alias` declaration with the same literal namespace URI and a higher import precedence.

[ERR XTSE0812] It is a static error if a value other than `#default` is specified for either the `stylesheet-prefix` or the `result-prefix` attributes of the `xsl:namespace-alias` element when there is no in-scope binding for that namespace prefix.

When a literal result element is processed, its namespace nodes are handled as follows:

- A namespace node whose string value is a literal namespace URI is not copied to the result tree.
- A namespace node whose string value is a target namespace URI is copied to the result tree, whether or not the URI identifies an excluded namespace.

In the event that the same URI is used as a literal namespace URI and a target namespace URI, the second of these rules takes precedence.

Note:

These rules achieve the effect that the element generated from the literal result element will have an in-scope namespace node that binds the `result-prefix` to the target namespace URI, provided that the namespace declaration associating this prefix with this URI is in scope for both the `xsl:namespace-alias` instruction and for the literal result element. Conversely, the `stylesheet-prefix` and the literal namespace URI will not normally appear in the result tree.

Example: Using `xsl:namespace-alias` to Generate a Stylesheet

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

For example, the stylesheet

```
<xsl:stylesheet
  version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fo="http://www.w3.org/1999/XSL/Format"
  xmlns:axsl="file://namespace.alias">

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

<xsl:template match="/">
  <axsl:stylesheet version="2.0">
    <xsl:apply-templates/>
  </axsl:stylesheet>
</xsl:template>

<xsl:template match="elements">
  <axsl:template match="/">
     <axsl:comment select="system-property('xsl:version')"/>
     <axsl:apply-templates/>
  </axsl:template>
</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>
```

The output of the transformation will be a stylesheet such as the following. Whitespace has been added for clarity. Note that an implementation may output different namespace prefixes from those appearing in this example; however, the rules guarantee that there will be a namespace node that binds the prefix `xsl` to the URI `http://www.w3.org/1999/XSL/Transform`, which makes it safe to use the QName `xsl:version` in the content of the generated stylesheet.

```
<xsl:stylesheet
  version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fo="http://www.w3.org/1999/XSL/Format">

<xsl:template match="/">
  <xsl:comment select="system-property('xsl:version')"/>
  <xsl:apply-templates/>
</xsl:template>

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

<xsl:template match="h1">
  <fo:block><xsl:apply-templates/></fo:block>
</xsl:template>

<xsl:template match="h2">
  <fo:block><xsl:apply-templates/></fo:block>
</xsl:template>

<xsl:template match="h3">
  <fo:block><xsl:apply-templates/></fo:block>
</xsl:template>

<xsl:template match="h4">
```

```
    <fo:block><xsl:apply-templates/></fo:block>
  </xsl:template>

  </xsl:stylesheet>
```

Note:

It may be necessary also to use aliases for namespaces other than the XSLT namespace URI. For example, it can be useful to define an alias for the namespace `http://www.w3.org/2001/XMLSchema-instance`, so that the stylesheet can use the attributes `xsi:type`, `xsi:nil`, and `xsi:schemaLocation` on a literal result element, without running the risk that a schema processor will interpret these as applying to the stylesheet itself. Equally, 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.

Example: Aliasing the XML Namespace

It is possible to define an alias for the XML namespace.

```
  <xsl:stylesheet xmlns:axml="http://www.example.com/alias-xml"
                  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                  version="2.0">

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

  <xsl:template match="/">
    <name axml:space="preserve">
      <first>James</first>
      <xsl:text> </xsl:text>
      <last>Clark</last>
    </name>
  </xsl:template>

  </xsl:stylesheet>
```

produces the output:

```
  <name xml:space="preserve"><first>James</first> <last>Clark</last></name>
```

This allows an `xml:space` attribute to be generated in the output without affecting the way the stylesheet is parsed. The same technique can be used for other attributes such as `xml:lang`, `xml:base`, and `xml:id`.

Note:

Namespace aliasing is only necessary when literal result elements are used. The problem of reserved namespaces does not arise when using `xsl:element` and `xsl:attribute` to construct the result tree. Therefore, as an alternative to using `xsl:namespace-alias`, it is always possible to achieve the desired effect by replacing literal result elements with `xsl:element` and `xsl:attribute` instructions.

11.2 Creating Element Nodes Using `xsl:element`

```
<!-- Category: instruction -->
<xsl:element
  name = { qname }
  namespace? = { uri-reference }
  inherit-namespaces? = "yes" | "no"
  use-attribute-sets? = qnames
  type? = qname
  validation? = "strict" | "lax" | "preserve" | "strip">
  <!-- Content: sequence-constructor -->
</xsl:element>
```

The `xsl:element` instruction allows an element to be created with a computed name. The expanded-QName of the element to be created is specified by a REQUIRED `name` attribute and an optional `namespace` attribute.

The content of the `xsl:element` instruction is a sequence constructor for the children, attributes, and namespaces of the created element. The sequence obtained by evaluating this sequence constructor (see *5.7 Sequence Constructors*) is used to construct the content of the element, as described in *5.7.1 Constructing Complex Content*.

The `xsl:element` element may have a `use-attribute-sets` attribute, whose value is a whitespace-separated list of QNames that identify `xsl:attribute-set` declarations. If this attribute is present, it is expanded as described in *10.2 Named Attribute Sets* to produce a sequence of attribute nodes. This sequence is prepended to the sequence produced as a result of evaluating the sequence constructor, as described in *5.7.1 Constructing Complex Content*.

The result of evaluating the `xsl:element` instruction, except in error cases, is the newly constructed element node.

The `name` attribute is interpreted as an attribute value template, whose effective value MUST be a lexical QName.

[ERR XTDE0820] It is a non-recoverable dynamic error if the effective value of the `name` attribute is not a lexical QName.

[ERR XTDE0830] In the case of an `xsl:element` instruction with no `namespace` attribute, it is a non-recoverable dynamic error if the effective value of the `name` attribute is a QName whose prefix is not declared in an in-scope namespace declaration for the `xsl:element` instruction.

If the `namespace` attribute is not present then the QName is expanded into an expanded-QName using the namespace declarations in effect for the `xsl:element` element, including any default namespace declaration.

If the `namespace` attribute is present, then it too is interpreted as an attribute value template. The effective value MUST be in the lexical space of the `xs:anyURI` type. If the string is zero-length, then the expanded-QName of the element has a null namespace URI. Otherwise, the string is used as the namespace URI of the expanded-QName of the element to be created. The local part of the lexical QName specified by the `name` attribute is used as the local part of the expanded-QName of the element to be created.

[ERR XTDE0835] It is a non-recoverable dynamic error if the effective value of the `namespace` attribute is not in the lexical space of the `xs:anyURI` data type.

Note:

The XDM data model requires the name of a node to be an instance of `xs:QName`, and XML Schema defines the namespace part of an `xs:QName` to be an instance of `xs:anyURI`. However, the schema specification, and the specifications that it refers to, give implementations some flexibility in how strictly they enforce these constraints.

The prefix of the lexical QName specified in the `name` attribute (or the absence of a prefix) is copied to the prefix part of the expanded-QName representing the name of the new element node. In the event of a conflict a prefix may subsequently be added, changed, or removed during the namespace fixup process (see *5.7.3 Namespace Fixup*).

The `xsl:element` instruction has an optional `inherit-namespaces` attribute, with the value `yes` or `no`. The default value is `yes`. If the value is set to `yes`, or is omitted, then the namespace nodes created for the newly constructed element (whether these were copied from those of the source node, or generated as a result of namespace fixup) are copied to the children and descendants of the newly constructed element, as described in *5.7.1 Constructing Complex Content*. If the value is set to `no`, then these namespace nodes are not automatically copied to the children. This may result in namespace undeclarations (such as `xmlns=""` or, in the case of XML Namespaces 1.1, `xmlns:p=""`) appearing on the child elements when a final result tree is serialized.

The base URI of the new element is copied from the base URI of the `xsl:element` instruction in the stylesheet, unless the content of the new element includes an `xml:base` attribute, in which case the base URI of the new element is the value of that attribute, resolved (if it is a relative URI) against the base URI of the `xsl:element` instruction in the stylesheet. (Note, however, that this is only relevant when creating parentless elements. When the new element is copied to form a child of an element or document node, the base URI of the new copy is taken from that of its new parent.)

11.2.1 Setting the Type Annotation for a Constructed Element Node

The optional attributes `type` and `validation` may be used on the `xsl:element` instruction to invoke validation of the contents of the element against a type definition or element declaration in a schema, and to determine the type annotation that the new element node will carry. These attributes also affect the type annotation carried by any elements and attributes that have the new element node as an ancestor. These two attributes are both optional, and if one is specified then the other MUST be omitted. The permitted values of these attributes and their semantics are described in *19.2 Validation*.

Note:

The final type annotation of the element in the result tree also depends on the `type` and `validation` attributes of the instructions used to create the ancestors of the element.

11.3 Creating Attribute Nodes Using `xsl:attribute`

```
<!-- Category: instruction -->
<xsl:attribute
  name = { qname }
  namespace? = { uri-reference }
  select? = expression
  separator? = { string }
  type? = qname
  validation? = "strict" | "lax" | "preserve" | "strip">
  <!-- Content: sequence-constructor -->
</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` or `xsl:copy`. The expanded-QName of the attribute to be created is specified by a REQUIRED `name` attribute and an optional `namespace` attribute. Except in error cases, the result of evaluating an `xsl:attribute` instruction is the newly constructed attribute node.

The string value of the new attribute node may be defined either by using the `select` attribute, or by the sequence constructor that forms the content of the `xsl:attribute` element. These are mutually exclusive. If neither is present, the value of the new attribute node will be a zero-length string. The way in which the value is constructed is specified in *5.7.2 Constructing Simple Content*.

[ERR XTSE0840] It is a static error if the `select` attribute of the `xsl:attribute` element is present unless the element has empty content.

If the `separator` attribute is present, then the effective value of this attribute is used to separate adjacent items in the result sequence, as described in *5.7.2 Constructing Simple Content*. In the absence of this attribute, the default separator is a single space (#x20) when the content is specified using the `select` attribute, or a zero-length string when the content is specified using a sequence constructor.

The `name` attribute is interpreted as an attribute value template, whose effective value MUST be a lexical QName.

[ERR XTDE0850] It is a non-recoverable dynamic error if the effective value of the `name` attribute is not a lexical QName.

[ERR XTDE0855] In the case of an `xsl:attribute` instruction with no `namespace` attribute, it is a non-recoverable dynamic error if the effective value of the `name` attribute is the string `xmlns`.

[ERR XTDE0860] In the case of an `xsl:attribute` instruction with no `namespace` attribute, it is a non-recoverable dynamic error if the effective value of the `name` attribute is a lexical QName whose prefix is not declared in an in-scope namespace declaration for the `xsl:attribute` instruction.

If the `namespace` attribute is not present, then the lexical QName is expanded into an expanded-QName 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 too is interpreted as an attribute value template. The effective value MUST be in the lexical space of the `xs:anyURI` type. If the string is zero-length, then the expanded-QName of the attribute has a null namespace URI. Otherwise, the string is used as the namespace URI of the expanded-QName of the attribute to be created. The local part of the lexical QName specified by the `name` attribute is used as the local part of the expanded-QName of the attribute to be created.

[ERR XTDE0865] It is a non-recoverable dynamic error if the effective value of the `namespace` attribute is not in the lexical space of the `xs:anyURI` data type.

Note:

The same considerations apply as for elements: [see ERR XTDE0835] in *11.2 Creating Element Nodes Using xsl:element* .

The prefix of the lexical QName specified in the `name` attribute (or the absence of a prefix) is copied to the prefix part of the expanded-QName representing the name of the new attribute node. In the event of a conflict this prefix (or absence of a prefix) may subsequently be changed during the namespace fixup process (see *5.7.3 Namespace Fixup*). If the attribute is in a non-null namespace and no prefix is specified, then the namespace fixup process will invent a prefix.

If the name of a constructed attribute is `xml:id`, the processor must perform attribute value normalization by effectively applying the `normalize-space`[FO] function to the value of the attribute, and the resulting attribute node must be given the `is-id` property. This applies whether the attribute is constructed using the `xsl:attribute` instruction or whether it is constructed using an attribute of a literal result element. This does not imply any constraints on the value of the attribute, or on its uniqueness, and it does not affect the type annotation of the attribute, unless the containing document is validated.

Note:

The effect of setting the `is-id` property is that the parent element can be located within the containing document by use of the `id`[FO] function. In effect, XSLT when constructing a document performs some of the functions of an `xml:id` processor, as defined in [xml:id]; the other aspects of `xml:id` processing are performed during validation.

Example: Creating a List-Valued Attribute

The following instruction creates the attribute `colors="red green blue"`:

```
<xsl:attribute name="colors" select="'red', 'green', 'blue'"/>
```

Example: Namespaces are not Attributes

It is not an error to write:

```
<xsl:attribute name="xmlns:xsl"
   namespace="file://some.namespace">http://www.w3.org/1999/XSL/Transform</xsl:attribute>
```

However, this will not result in the namespace declaration `xmlns:xsl="http://www.w3.org/1999/XSL/Transform"` being output. Instead, it will produce an attribute node with local name `xsl`, and with a system-allocated namespace prefix mapped to the namespace URI `file://some.namespace`. This is because the namespace fixup process is not allowed to use `xmlns` as the name of a namespace node.

As described in *5.7.1 Constructing Complex Content*, in a sequence that is used to construct the content of an element, any attribute nodes MUST appear in the sequence before any element, text, comment, or processing instruction nodes. Where the sequence contains two or more attribute nodes with the same expanded-QName, the one that comes last is the only one that takes effect.

Note:

If a collection of attributes is generated repeatedly, this can be done conveniently by using named attribute sets: see *10.2 Named Attribute Sets*

11.3.1 Setting the Type Annotation for a Constructed Attribute Node

The optional attributes `type` and `validation` may be used on the `xsl:attribute` instruction to invoke validation of the contents of the attribute against a type definition or attribute declaration in a schema, and to determine the type annotation that the new attribute node will carry. These two attributes are both optional, and if one is specified then the other MUST be omitted. The permitted values of these attributes and their semantics are described in *19.2 Validation*.

Note:

The final type annotation of the attribute in the result tree also depends on the `type` and `validation` attributes of the instructions used to create the ancestors of the attribute.

11.4 Creating Text Nodes

This section describes three different ways of creating text nodes: by means of literal text nodes in the stylesheet, or by using the `xsl:text` and `xsl:value-of` instructions. It is also possible to create text nodes using the `xsl:number` instruction described in *12 Numbering*.

If and when the sequence that results from evaluating a sequence constructor is used to form the content of a node, as described in *5.7.2 Constructing Simple Content* and *5.7.1 Constructing Complex Content*, adjacent text nodes in the sequence are merged. Within the sequence itself, however, they exist as distinct nodes.

Example: A sequence of text nodes

The following function returns a sequence of three text nodes:

```
<xsl:function name="f:wrap">
  <xsl:param name="s"/>
  <xsl:text>(</xsl:text>
  <xsl:value-of select="$s"/>
  <xsl:text>)</xsl:text>
</xsl:function>
```

When this function is called as follows:

```
<xsl:value-of select="f:wrap('---')"/>
```

the result is:

```
(---)
```

No additional spaces are inserted, because the calling xsl:value-of instruction merges adjacent text nodes before atomizing the sequence. However, the result of the instruction:

```
<xsl:value-of select="data(f:wrap('---'))"/>
```

is:

```
( --- )
```

because in this case the three text nodes are atomized to form three strings, and spaces are inserted between adjacent strings.

It is possible to construct text nodes whose string value is zero-length. A zero-length text node, when atomized, produces a zero-length string. However, zero-length text nodes are ignored when they appear in a sequence that is used to form the content of a node, as described in *5.7.1 Constructing Complex Content* and *5.7.2 Constructing Simple Content*.

11.4.1 Literal Text Nodes

A sequence constructor can contain text nodes. Each text node in a sequence constructor remaining after whitespace text nodes have been stripped as specified in *4.2 Stripping Whitespace from the Stylesheet* will construct a new text node with the same string value. The resulting text node is added to the result of the containing sequence constructor.

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 serialized as an XML document, unless otherwise specified using character maps (see *20.1 Character Maps*) or `disable-output-escaping` (see *20.2 Disabling Output Escaping*).

11.4.2 Creating Text Nodes Using `xsl:text`

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

The xsl:text element is evaluated to contruct a new text node. The content of the xsl:text element is a single text node whose value forms the string value of the new text node. An xsl:text element may be empty, in which case the result of evaluating the instruction is a text node whose string value is the zero-length string.

The result of evaluating an xsl:text instruction is the newly constructed text node.

A text node that is an immediate child of an xsl:text instruction will not be stripped from the stylesheet tree, even if it consists entirely of whitespace (see *4.4 Stripping Whitespace from a Source Tree*).

For the effect of the deprecated `disable-output-escaping` attribute, see *20.2 Disabling Output Escaping*

> **Note:**
>
> It is not always necessary to use the xsl:text instruction to write text nodes to the result tree. Literal text can be written to the result tree by including it anywhere in a sequence constructor, while computed text can be output using the xsl:value-of instruction. The principal reason for using xsl:text is that it offers improved control over whitespace handling.

11.4.3 Generating Text with `xsl:value-of`

Within a sequence constructor, the xsl:value-of instruction can be used to generate computed text nodes. The xsl:value-of instruction computes the text using an expression that is specified as the value of the `select` attribute, or by means of contained instructions. This might, for example, extract text from a source tree or insert the value of a variable.

```
<!-- Category: instruction -->
<xsl:value-of
  select? = expression
  separator? = { string }
```

```
  [disable-output-escaping]? = "yes" | "no">
  <!-- Content: sequence-constructor -->
</xsl:value-of>
```

The xsl:value-of instruction is evaluated to construct a new text node; the result of the instruction is the newly constructed text node.

The string value of the new text node may be defined either by using the select attribute, or by the sequence constructor (see *5.7 Sequence Constructors*) that forms the content of the xsl:value-of element. These are mutually exclusive, and one of them must be present. The way in which the value is constructed is specified in *5.7.2 Constructing Simple Content*.

[ERR XTSE0870] It is a static error if the select attribute of the xsl:value-of element is present when the content of the element is non-empty, or if the select attribute is absent when the content is empty.

If the separator attribute is present, then the effective value of this attribute is used to separate adjacent items in the result sequence, as described in *5.7.2 Constructing Simple Content*. In the absence of this attribute, the default separator is a single space (#x20) when the content is specified using the select attribute, or a zero-length string when the content is specified using a sequence constructor.

Special rules apply when backwards compatible behavior is enabled for the instruction. If no separator attribute is present, and if the select attribute is present, then all items in the atomized result sequence other than the first are ignored.

Example: Generating a List with Separators

The instruction:

```
<x><xsl:value-of select="1 to 4" separator="|"/></x>
```

produces the output:

```
<x>1|2|3|4</x>
```

Note:

The xsl:copy-of element can be used to copy a sequence of nodes to the result tree without atomization. See *11.9.2 Deep Copy*.

For the effect of the deprecated disable-output-escaping attribute, see *20.2 Disabling Output Escaping*

11.5 Creating Document Nodes

```
<!-- Category: instruction -->
<xsl:document
  validation? = "strict" | "lax" | "preserve" | "strip"
  type? = qname>
  <!-- Content: sequence-constructor -->
</xsl:document>
```

The xsl:document instruction is used to create a new document node. The content of the xsl:document element is a sequence constructor for the children of the new document node. A document node is created, and the sequence obtained by evaluating the sequence constructor is used to construct the content of the document, as described in *5.7.1 Constructing Complex Content*. The temporary tree rooted at this document node forms the result tree.

Except in error situations, the result of evaluating the xsl:document instruction is a single node, the newly constructed document node.

Note:

The new document is not serialized. To construct a document that is to form a final result rather than an intermediate result, use the xsl:result-document instruction described in *19.1 Creating Final Result Trees*.

The optional attributes type and validation may be used on the xsl:document instruction to validate the contents of the new document, and to determine the type annotation that elements and attributes within the result tree will carry. The permitted values and their semantics are described in *19.2.2 Validating Document Nodes*.

The base URI of the new document node is taken from the base URI of the xsl:document instruction.

The document-uri and unparsed-entities properties of the new document node are set to empty.

Example: Checking Uniqueness Constraints in a Temporary Tree

The following example creates a temporary tree held in a variable. The use of an enclosed xsl:document instruction ensures that uniqueness constraints defined in the schema for the relevant elements are checked.

```
<xsl:variable name="tree" as="document-node()">
  <xsl:document validation="strict">
    <xsl:apply-templates/>
  </xsl:document>
</xsl:variable>
```

11.6 Creating Processing Instructions

```
<!-- Category: instruction -->
<xsl:processing-instruction
```

```
<!-- Category: instruction -->
<xsl:processing-instruction
  name = { ncname }
  select? = expression>
  <!-- Content: sequence-constructor -->
</xsl:processing-instruction>
```

The xsl:processing-instruction element is evaluated to create a 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.

The string value of the new processing-instruction node may be defined either by using the select attribute, or by the sequence constructor that forms the content of the xsl:processing-instruction element. These are mutually exclusive. If neither is present, the string value of the new processing-instruction node will be a zero-length string. The way in which the value is constructed is specified in *5.7.2 Constructing Simple Content*.

[ERR XTSE0880] It is a static error if the select attribute of the xsl:processing-instruction element is present unless the element has empty content.

Except in error situations, the result of evaluating the xsl:processing-instruction instruction is a single node, the newly constructed processing instruction node.

Example: Creating a Processing Instruction

This instruction:

```
<xsl:processing-instruction name="xml-stylesheet"
  select="('href=&quot;book.css&quot;', 'type=&quot;text/css&quot;)"/>
```

creates the processing instruction

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

Note that the xml-stylesheet processing instruction contains *pseudo-attributes* in the form name="value". Although these have the same textual form as attributes in an element start tag, they are not represented as XDM attribute nodes, and cannot therefore be constructed using xsl:attribute instructions.

[ERR XTDE0890] It is a non-recoverable dynamic error if the effective value of the name attribute is not both an NCName [Names] and a PITarget [XML].

Note:

Because these rules disallow the name xml, the xsl:processing-instruction cannot be used to output an XML declaration. The xsl:output declaration should be used to control this instead (see *20 Serialization*).

If the result of evaluating the content of the xsl:processing-instruction contains the string ?>, this string is modified by inserting a space between the ? and > characters.

The base URI of the new processing-instruction is copied from the base URI of the xsl:processing-instruction element in the stylesheet. (Note, however, that this is only relevant when creating a parentless processing instruction. When the new processing instruction is copied to form a child of an element or document node, the base URI of the new copy is taken from that of its new parent.)

11.7 Creating Namespace Nodes

```
<!-- Category: instruction -->
<xsl:namespace
  name = { ncname }
  select? = expression>
  <!-- Content: sequence-constructor -->
</xsl:namespace>
```

The xsl:namespace element is evaluated to create a namespace node. Except in error situations, the result of evaluating the xsl:namespace instruction is a single node, the newly constructed namespace node.

The xsl:namespace element has a REQUIRED name attribute that specifies the name of the namespace node (that is, the namespace prefix). The value of the name attribute is interpreted as an attribute value template. If the effective value of the name attribute is a zero-length string, a namespace node is added for the default namespace.

The string value of the new namespace node (that is, the namespace URI) may be defined either by using the select attribute, or by the sequence constructor that forms the content of the xsl:namespace element. These are mutually exclusive. Since the string value of a namespace node cannot be a zero-length string, one of them must be present. The way in which the value is constructed is specified in *5.7.2 Constructing Simple Content*.

[ERR XTDE0905] It is a non-recoverable dynamic error if the string value of the new namespace node is not valid in the lexical space of the data type xs:anyURI. [see ERR XTDE0835]

[ERR XTSE0910] It is a static error if the select attribute of the xsl:namespace element is present when the element has content other than one or more xsl:fallback instructions, or if the select attribute is absent when the element has empty content.

Note the restrictions described in *5.7.1 Constructing Complex Content* for the position of a namespace node relative to other nodes in the node sequence returned by a sequence constructor.

Example: Constructing a QName-Valued Attribute

This literal result element:

```
<data xsi:type="xs:integer" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <xsl:namespace name="xs" select="'http://www.w3.org/2001/XMLSchema'"/>
  <xsl:text>42</xsl:text>
</data>
```

would typically cause the output document to contain the element:

```
<data xsi:type="xs:integer"
     xmlns:xs="http://www.w3.org/2001/XMLSchema"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">42</data>
```

In this case, the element is constructed using a literal result element, and the namespace `xmlns:xs="http://www.w3.org/2001/XMLSchema"` could therefore have been added to the result tree simply by declaring it as one of the in-scope namespaces in the stylesheet. In practice, the `xsl:namespace` instruction is more likely to be useful in situations where the element is constructed using an `xsl:element` instruction, which does not copy all the in-scope namespaces from the stylesheet.

[ERR XTDE0920] It is a non-recoverable dynamic error if the effective value of the `name` attribute is neither a zero-length string nor an NCName [Names], or if it is `xmlns`.

[ERR XTDE0925] It is a non-recoverable dynamic error if the `xsl:namespace` instruction generates a namespace node whose name is `xml` and whose string value is not `http://www.w3.org/XML/1998/namespace`, or a namespace node whose string value is `http://www.w3.org/XML/1998/namespace` and whose name is not `xml`.

[ERR XTDE0930] It is a non-recoverable dynamic error if evaluating the `select` attribute or the contained sequence constructor of an `xsl:namespace` instruction results in a zero-length string.

For details of other error conditions that may arise, see *5.7 Sequence Constructors*.

Note:

It is rarely necessary to use `xsl:namespace` to create a namespace node in the result tree; in most circumstances, the required namespace nodes will be created automatically, as a side-effect of writing elements or attributes that use the namespace. An example where `xsl:namespace` is needed is a situation where the required namespace is used only within attribute values in the result document, not in element or attribute names; especially where the required namespace prefix or namespace URI is computed at run-time and is not present in either the source document or the stylesheet.

Adding a namespace node to the result tree will never change the expanded-QName of any element or attribute node in the result tree: that is, it will never change the namespace URI of an element or attribute. It might, however, constrain the choice of prefixes when namespace fixup is performed.

Namespace prefixes for element and attribute names are effectively established by the namespace fixup process described in *5.7.3 Namespace Fixup*. The fixup process ensures that an element has in-scope namespace nodes for the namespace URIs used in the element name and in its attribute names, and the serializer will typically use these namespace nodes to determine the prefix to use in the serialized output. The fixup process cannot generate namespace nodes that are inconsistent with those already present in the tree. This means that it is not possible for the processor to decide the prefix to use for an element or for any of its attributes until all the namespace nodes for the element have been added.

If a namespace prefix is mapped to a particular namespace URI using the `xsl:namespace` instruction, or by using `xsl:copy` or `xsl:copy-of` to copy a namespace node, this prevents the namespace fixup process (and hence the serializer) from using the same prefix for a different namespace URI on the same element.

Example: Conflicting Namespace Prefixes

Given the instruction:

```
<xsl:element name="p:item" xmlns:p="http://www.example.com/p">
  <xsl:namespace name="p">http://www.example.com/q</xsl:namespace>
</xsl:element>
```

a possible serialization of the result tree is:

```
<ns0:item xmlns:ns0="http://www.example.com/p" xmlns:p="http://www.example.com/q"/>
```

The processor must invent a namespace prefix for the URI `p.uri`; it cannot use the prefix `p` because that prefix has been explicitly associated with a different URI.

Note:

The `xsl:namespace` instruction cannot be used to generate a **namespace undeclaration** of the form `xmlns=""` (nor the new forms of namespace undeclaration permitted in [Namespaces in XML 1.1]). Namespace undeclarations are generated automatically by the serializer if `undeclare-prefixes="yes"` is specified on `xsl:output`, whenever a parent element has a namespace node for the default namespace prefix, and a child element has no namespace node for that prefix.

11.8 Creating Comments

```
<!-- Category: instruction -->
<xsl:comment
  select? = expression>
  <!-- Content: sequence-constructor -->
</xsl:comment>
```

The xsl:comment element is evaluated to contruct a new comment node. Except in error cases, the result of evaluating the xsl:comment instruction is a single node, the newly constructed comment node.

The string value of the new comment node may be defined either by using the select attribute, or by the sequence constructor that forms the content of the xsl:comment element. These are mutually exclusive. If neither is present, the value of the new comment node will be a zero-length string. The way in which the value is constructed is specified in *5.7.2 Constructing Simple Content*.

[ERR XTSE0940] It is a static error if the select attribute of the xsl:comment element is present unless the element has empty content.

Example: Generating a 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!-->
```

In the generated comment node, the processor MUST insert a space after any occurrence of - that is followed by another - or that ends the comment.

11.9 Copying Nodes

11.9.1 Shallow Copy

```
<!-- Category: instruction -->
<xsl:copy
  copy-namespaces? = "yes" | "no"
  inherit-namespaces? = "yes" | "no"
  use-attribute-sets? = qnames
  type? = qname
  validation? = "strict" | "lax" | "preserve" | "strip">
  <!-- Content: sequence-constructor -->
</xsl:copy>
```

The xsl:copy instruction provides a way of copying the context item. If the context item is a node, evaluating the xsl:copy instruction constructs a copy of the context node, and the result of the xsl:copy instruction is this newly constructed node. By default, the namespace nodes of the context node are automatically copied as well, but the attributes and children of the node are not automatically copied.

When the context item is an atomic value, the xsl:copy instruction returns this value. The sequence constructor, if present, is not evaluated.

When the context item is an attribute node, text node, comment node, processing instruction node, or namespace node, the xsl:copy instruction returns a new node that is a copy of the context node. The new node will have the same node kind, name, and string value as the context node. In the case of an attribute node, it will also have the same values for the is-id and is-idrefs properties. The sequence constructor, if present, is not evaluated.

When the context item is a document node or element node, the xsl:copy instruction returns a new node that has the same node kind and name as the context node. The content of the new node is formed by evaluating the sequence constructor contained in the xsl:copy instruction. The sequence obtained by evaluating this sequence constructor is used (after prepending any attribute nodes or namespace nodes as described in the following paragraphs) to construct the content of the document or element node, as described in *5.7.1 Constructing Complex Content*.

Example: Identity Transformation

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>
```

This template rule can be used to copy any node in a tree by applying template rules to its attributes and children. It can be combined with additional template rules that modify selected nodes, for example if all nodes are to be copied except note elements and their contents, this can be achieved by using the identity template rule together with the template rule:

```
<xsl:template match="note"/>
```

Note:

The xsl:copy instruction is most useful when copying element nodes. In other cases, the xsl:copy-of instruction is more flexible,

because it has a `select` attribute allowing selection of the nodes or values to be copied.

The `xsl:copy` instruction has an optional `use-attribute-sets` attribute, whose value is a whitespace-separated list of QNames that identify `xsl:attribute-set` declarations. This attribute is used only when copying element nodes. This list is expanded as described in *10.2 Named Attribute Sets* to produce a sequence of attribute nodes. This sequence is prepended to the sequence produced as a result of evaluating the sequence constructor.

The `xsl:copy` instruction has an optional `copy-namespaces` attribute, with the value `yes` or `no`. The default value is `yes`. The attribute is used only when copying element nodes. If the value is set to `yes`, or is omitted, then all the namespace nodes of the source element are copied as namespace nodes for the result element. These copied namespace nodes are prepended to the sequence produced as a result of evaluating the sequence constructor (it is immaterial whether they come before or after any attribute nodes produced by expanding the `use-attribute-sets` attribute). If the value is set to `no`, then the namespace nodes are not copied. However, namespace nodes will still be added to the result element as REQUIRED by the namespace fixup process: see *5.7.3 Namespace Fixup*.

The `xsl:copy` instruction has an optional `inherit-namespaces` attribute, with the value `yes` or `no`. The default value is `yes`. The attribute is used only when copying element nodes. If the value is set to `yes`, or is omitted, then the namespace nodes created for the newly constructed element (whether these were copied from those of the source node, or generated as a result of namespace fixup) are copied to the children and descendants of the newly constructed element, as described in *5.7.1 Constructing Complex Content*. If the value is set to `no`, then these namespace nodes are not automatically copied to the children. This may result in namespace undeclarations (such as `xmlns=""` or, in the case of XML Namespaces 1.1, `xmlns:p=""`) appearing on the child elements when a final result tree is serialized.

[ERR XTTE0950] It is a type error to use the `xsl:copy` or `xsl:copy-of` instruction to copy a node that has namespace-sensitive content if the `copy-namespaces` attribute has the value `no` and its explicit or implicit `validation` attribute has the value `preserve`. It is also a type error if either of these instructions (with `validation="preserve"`) is used to copy an attribute having namespace-sensitive content, unless the parent element is also copied. A node has namespace-sensitive content if its typed value contains an item of type `xs:QName` or `xs:NOTATION` or a type derived therefrom. The reason this is an error is because the validity of the content depends on the namespace context being preserved.

Note:

When attribute nodes are copied, whether with `xsl:copy` or with `xsl:copy-of`, the processor does not automatically copy any associated namespace information. The namespace used in the attribute name itself will be declared by virtue of the namespace fixup process (see *5.7.3 Namespace Fixup*) when the attribute is added to an element in the result tree, but if namespace prefixes are used in the content of the attribute (for example, if the value of the attribute is an XPath expression) then it is the responsibility of the stylesheet author to ensure that suitable namespace nodes are added to the result tree. This can be achieved by copying the namespace nodes using `xsl:copy`, or by generating them using `xsl:namespace`.

The optional attributes `type` and `validation` may be used on the `xsl:copy` instruction to validate the contents of an element, attribute or document node against a type definition, element declaration, or attribute declaration in a schema, and thus to determine the type annotation that the new copy of an element or attribute node will carry. These attributes are ignored when copying an item that is not an element, attribute or document node. When the node being copied is an element or document node, these attributes also affect the type annotation carried by any elements and attributes that have the copied element or document node as an ancestor. These two attributes are both optional, and if one is specified then the other MUST be omitted. The permitted values of these attributes and their semantics are described in *19.2 Validation*.

Note:

The final type annotation of the node in the result tree also depends on the `type` and `validation` attributes of the instructions used to create the ancestors of the node.

The base URI of a node is copied, except in the case of an element node having an `xml:base` attribute, in which case the base URI of the new node is taken as the value of the `xml:base` attribute, resolved if it is relative against the base URI of the `xsl:copy` instruction. If the copied node is subsequently attached as a child to a new element or document node, the final copy of the node inherits its base URI from its parent node, unless this is overridden using an `xml:base` attribute.

When an `xml:id` attribute is copied, using either the `xsl:copy` or `xsl:copy-of` instruction, it is implementation-defined whether the value of the attribute is subjected to attribute value normalization (that is, effectively applying the `normalize-space`[FO] function).

Note:

In most cases the value will already have been subjected to attribute value normalization on the source tree, but if this processing has not been performed on the source tree, it is not an error for it to be performed on the result tree.

11.9.2 Deep Copy

```
<!-- Category: instruction -->
<xsl:copy-of
  select = expression
  copy-namespaces? = "yes" | "no"
  type? = qname
  validation? = "strict" | "lax" | "preserve" | "strip" />
```

The `xsl:copy-of` instruction can be used to construct a copy of a sequence of nodes and/or atomic values, with each new node containing copies of all the children, attributes, and (by default) namespaces of the original node, recursively. The result of evaluating the instruction is a sequence of items corresponding one-to-one with the supplied sequence, and retaining its order.

The REQUIRED `select` attribute contains an expression, whose value may be any sequence of nodes and atomic values. The items in this sequence are processed as follows:

- If the item is an element node, a new element is constructed and appended to the result sequence. The new element will have the same expanded-QName as the original, and it will have deep copies of the attribute nodes and children of the element node.

  The new element will also have namespace nodes copied from the original element node, unless they are excluded by specifying `copy-namespaces="no"`. If this attribute is omitted, or takes the value `yes`, then all the namespace nodes of the original element are copied to the new element. If it takes the value `no`, then none of the namespace nodes are copied: however, namespace nodes will still be created in the result tree as REQUIRED by the namespace fixup process: see *5.7.3 Namespace Fixup*. This attribute affects all

elements copied by this instruction: both elements selected directly by the `select` expression, and elements that are descendants of nodes selected by the `select` expression.

The new element will have the same values of the `is-id`, `is-idrefs`, and `nilled` properties as the original element.

- If the item is a document node, the instruction adds a new document node to the result sequence; the children of this document node will be one-to-one copies of the children of the original document node (each copied according to the rules for its own node kind).
- If the item is an attribute or namespace node, or a text node, a comment, or a processing instruction, the same rules apply as with `xsl:copy` (see *11.9.1 Shallow Copy*).
- If the item is an atomic value, the value is appended to the result sequence, as with `xsl:sequence`.

The optional attributes `type` and `validation` may be used on the `xsl:copy-of` instruction to validate the contents of an element, attribute or document node against a type definition, element declaration, or attribute declaration in a schema and thus to determine the type annotation that the new copy of an element or attribute node will carry. These attributes are applied individually to each element, attribute, and document node that is selected by the expression in the `select` attribute. These attributes are ignored when copying an item that is not an element, attribute or document node.

The specified `type` and `validation` apply directly only to elements, attributes and document nodes created as copies of nodes actually selected by the `select` expression, they do not apply to nodes that are implicitly copied because they have selected nodes as an ancestor. However, these attributes do indirectly affect the type annotation carried by such implicitly copied nodes, as a consequence of the validation process.

These two attributes are both optional, and if one is specified then the other MUST be omitted. The permitted values of these attributes and their semantics are described in *19.2 Validation*.

Errors may occur when copying namespace-sensitive elements or attributes using `validation="preserve"`. [see ERR XTTE0950].

The base URI of a node is copied, except in the case of an element node having an `xml:base` attribute, in which case the base URI of the new node is taken as the value of the `xml:base` attribute, resolved if it is relative against the base URI of the `xsl:copy-of` instruction. If the copied node is subsequently attached as a child to a new element or document node, the final copy of the node inherits its base URI from its parent node, unless this is overridden using an `xml:base` attribute.

11.10 Constructing Sequences

```
<!-- Category: instruction -->
<xsl:sequence
  select = expression>
  <!-- Content: xsl:fallback* -->
</xsl:sequence>
```

The `xsl:sequence` instruction may be used within a sequence constructor to construct a sequence of nodes and/or atomic values. This sequence is returned as the result of the instruction. Unlike most other instructions, `xsl:sequence` can return a sequence containing existing nodes, rather than constructing new nodes. When `xsl:sequence` is used to add atomic values to a sequence, the effect is very similar to the `xsl:copy-of` instruction.

The items comprising the result sequence are selected using the `select` attribute.

Any contained `xsl:fallback` instructions are ignored by an XSLT 2.0 processor, but can be used to define fallback behavior for an XSLT 1.0 processor running in forwards compatibility mode.

Example: Constructing a Sequence of Integers

For example, the following code:

```
<xsl:variable name="values" as="xs:integer*">
    <xsl:sequence select="(1,2,3,4)"/>
    <xsl:sequence select="(8,9,10)"/>
</xsl:variable>
<xsl:value-of select="sum($values)"/>
```

produces the output: `37`

Example: Using `xsl:for-each` to Construct a Sequence

The following code constructs a sequence containing the value of the `@price` attribute for selected elements (which we assume to be typed as `xs:decimal`), or a computed price for those elements that have no `@price` attribute. It then returns the average price:

```
<xsl:variable name="prices" as="xs:decimal*">
  <xsl:for-each select="//product">
    <xsl:choose>
      <xsl:when test="@price">
        <xsl:sequence select="@price"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:sequence select="@cost * 1.5"/>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:for-each>
</xsl:variable>
<xsl:value-of select="avg($prices)"/>
```

Note that the existing `@price` attributes could equally have been added to the `$prices` sequence using `xsl:copy-of` or `xsl:value-of`. However, `xsl:copy-of` would create a copy of the attribute node, which is not needed in this situation, while `xsl:value-of` would create a new text node, which then has to be converted to an `xs:decimal`. Using `xsl:sequence`, which in this case atomizes the existing attribute node and adds an `xs:decimal` atomic value to the result sequence, is a more direct way of achieving the same result.

This example could alternatively be solved at the XPath level:

```
<xsl:value-of select="avg(//product/(+@price, @cost*1.5)[1])"/>
```

(The apparently redundant `+` operator is there to atomize the attribute value: the expression on the right hand side of the `/` operator must not return a mixture of nodes and atomic values.)

12 Numbering

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

The `xsl:number` instruction is used to create a formatted number. The result of the instruction is a newly constructed text node containing the formatted number as its string value.

[DEFINITION: The `xsl:number` instruction performs two tasks: firstly, determining a **place marker** (this is a sequence of integers, to allow for hierarchic numbering schemes such as `1.12.2` or `3(c)ii`), and secondly, formatting the place marker for output as a text node in the result sequence.] The place marker to be formatted can either be supplied directly, in the `value` attribute, or it can be computed based on the position of a selected node within the tree that contains it.

[ERR XTSE0975] It is a static error if the `value` attribute of `xsl:number` is present unless the `select`, `level`, `count`, and `from` attributes are all absent.

Note:

The facilities described in this section are specifically designed to enable the calculation and formatting of section numbers, paragraph numbers, and the like. For formatting of other numeric quantities, the `format-number` function may be more suitable: see *16.4 Number Formatting*.

12.1 Formatting a Supplied Number

The place marker to be formatted may be specified by an expression. The `value` attribute contains the expression. The value of this expression is atomized using the procedure defined in [XPath 2.0], and each value *$V* in the atomized sequence is then converted to the integer value returned by the XPath expression `xs:integer(round(number($V)))`. The resulting sequence of integers is used as the place marker to be formatted.

If backwards compatible behavior is enabled for the instruction, then:

- all items in the atomized sequence after the first are discarded;
- If the atomized sequence is empty, it is replaced by a sequence containing the `xs:double` value `NaN` as its only item;
- If any value in the sequence cannot be converted to an integer (this includes the case where the sequence contains a `NaN` value) then the string `NaN` is inserted into the formatted result string in its proper position. The error described in the following paragraph does not apply in this case.

[ERR XTDE0980] It is a non-recoverable dynamic error if any undiscarded item in the atomized sequence supplied as the value of the `value` attribute of `xsl:number` cannot be converted to an integer, or if the resulting integer is less than 0 (zero).

Note:

The value zero does not arise when numbering nodes in a source document, but it can arise in other numbering sequences. It is permitted specifically because the rules of the `xsl:number` instruction are also invoked by functions such as `format-time`: the minutes and seconds component of a time value can legitimately be zero.

The resulting sequence is formatted as a string using the effective values of the attributes specified in *12.3 Number to String Conversion Attributes*; each of these attributes is interpreted as an attribute value template. After conversion, the `xsl:number` element constructs a new text node containing the resulting string, and returns this node.

Example: Numbering a Sorted List

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>
```

12.2 Numbering based on Position in a Document

If no `value` attribute is specified, then the `xsl:number` instruction returns a new text node containing a formatted place marker that is based on the position of a selected node within its containing document. If the `select` attribute is present, then the expression contained in the `select` attribute is evaluated to determine the selected node. If the `select` attribute is omitted, then the selected node is the context node.

[ERR XTTE0990] It is a type error if the `xsl:number` instruction is evaluated, with no `value` or `select` attribute, when the context item is not a node.

[ERR XTTE1000] It is a type error if the result of evaluating the `select` attribute of the `xsl:number` instruction is anything other than a single node.

The following attributes control how the selected node is to be numbered:

- The `level` attribute specifies rules for selecting the nodes that are taken into account in allocating a number; it has the values `single`, `multiple` or `any`. The default is `single`.
- The `count` attribute is a pattern that specifies which nodes are to 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 kind as the selected node and, if the selected node has an expanded-QName, with the same expanded-QName as the selected node.
- The `from` attribute is a pattern that specifies where counting starts.

In addition, the attributes specified in *12.3 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 sequence of positive integers using the `level`, `count` and `from` attributes. Where `level` is `single` or `any`, this sequence will either be empty or contain a single number; where `level` is `multiple`, the sequence may be of any length. The sequence is constructed as follows:

Let `matches-count($node)` be a function that returns true if and only if the given node `$node` matches the pattern given in the `count` attribute, or the implied pattern (according to the rules given above) if the `count` attribute is omitted.

Let `matches-from($node)` be a function that returns true if and only if the given node `$node` matches the pattern given in the `from` attribute, or if `$node` is the root node of a tree. If the `from` attribute is omitted, then the function returns true if and only if `$node` is the root node of a tree.

Let `$S` be the selected node.

When `level="single"`:

- Let `$A` be the node sequence selected by the following expression:
  ```
  $S/ancestor-or-self::node()[matches-count(.)][1]
  ```
  (this selects the innermost ancestor-or-self node that matches the `count` pattern)
- Let `$F` be the node sequence selected by the expression
  ```
  $S/ancestor-or-self::node()[matches-from(.)][1]
  ```
  (this selects the innermost ancestor-or-self node that matches the `from` pattern):
- Let `$AF` be the value of:
  ```
  $A[ancestor-or-self::node()[. is $F]]
  ```
  (this selects $A if it is in the subtree rooted at $F, or the empty sequence otherwise)
- If `$AF` is empty, return the empty sequence, `()`
- Otherwise return the value of:
  ```
  1 + count($AF/preceding-sibling::node()[matches-count(.)])
  ```
  (the number of preceding siblings of the counted node that match the `count` pattern, plus one).

When `level="multiple"`:

- Let `$A` be the node sequence selected by the expression
  ```
  $S/ancestor-or-self::node()[matches-count(.)]
  ```
  (the set of ancestor-or-self nodes that match the `count` pattern)
- Let `$F` be the node sequence selected by the expression
  ```
  $S/ancestor-or-self::node()[matches-from(.)][1]
  ```
  (the innermost ancestor-or-self node that matches the `from` pattern)
- Let `$AF` be the value of
  ```
  $A[ancestor-or-self::node()[. is $F]]
  ```
  (the nodes selected in the first step that are in the subtree rooted at the node selected in the second step)
- Return the result of the expression
  ```
  for $af in $AF return 1+count($af/preceding-sibling::node()[matches-count(.)])
  ```
  (a sequence of integers containing, for each of these nodes, one plus the number of preceding siblings that match the `count` pattern)

When `level="any"`:

- Let `$A` be the node sequence selected by the expression
  ```
  $S/(preceding::node()|ancestor-or-self::node())[matches-count(.)]
  ```
  (the set of nodes consisting of the selected node together with all nodes, other than attributes and namespaces, that precede the selected node in document order, provided that they match the `count` pattern)

- Let `$F` be the node sequence selected by the expression
  `$S/(preceding::node()|ancestor::node())[matches-from(.)][last()]`
  (the last node in document order that matches the `from` pattern and that precedes the selected node, using the same definition)
- Let `$AF` be the node sequence `$A[. is $F or . >> $F]`.
  (the nodes selected in the first step, excluding those that precede the node selected in the second step)
- If `$AF` is empty, return the empty sequence, `()`
- Otherwise return the value of the expression `count($AF)`

The sequence of numbers (the place marker) is then converted into a string using the effective values of the attributes specified in *12.3 Number to String Conversion Attributes*; each of these attributes is interpreted as an attribute value template. After conversion, the resulting string is used to create a text node, which forms the result of the `xsl:number` instruction.

Example: Numbering the Items in an Ordered List

The following will 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>
```

Example: Multi-Level Numbering

The following two rules will 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. Subsections within a chapter are numbered 1.1.1, 1.1.2, 1.1.3; subsections within an appendix are numbered A.1.1, A.1.2, A.1.3.

```
<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>
```

Example: Numbering Notes within a Chapter

This 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>
```

12.3 Number to String Conversion Attributes

The following attributes are used to control conversion of a sequence of numbers into a string. The numbers are integers greater than or equal to 0 (zero). 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*) indicate the format to be used for each number in the sequence; in most cases the format token is the same as the required representation of the number 1 (one).

Each non-alphanumeric token is either a prefix, a separator, or a suffix. If there is a non-alphanumeric token but no format token, then the single non-alphanumeric token is used as both the prefix and the suffix. The prefix, if it exists, is the non-alphanumeric token that precedes the first format token: the prefix always appears exactly once in the constructed string, at the start. The suffix, if it exists, is the non-

alphanumeric token that follows the last format token: the suffix always appears exactly once in the constructed string, at the end. All other non-alphanumeric tokens (those that occur between two format tokens) are *separator tokens* and are used to separate numbers in the sequence.

The *n*th format token is used to format the *n*th number in the sequence. If there are more numbers than format tokens, then the last format token is used to format remaining numbers. If there are no format tokens, then a format token of `1` is used to format all numbers. Each number after the first is separated from the preceding number by the separator token preceding the format token used to format that number, or, if that is the first format token, then by `.` (dot).

Example: Formatting a List of Numbers

Given the sequence of numbers `5, 13, 7` and the format token `A-001(i)`, the output will be the string `E-013(vii)`

Format tokens 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. The digits used in the decimal representation are the set of digits containing the digit character used in the format token. Thus, a format token `1` generates the sequence `0 1 2 ... 10 11 12 ...`, and a format token `01` generates the sequence `00 01 02 ... 09 10 11 12 ... 99 100 101`. A format token of `&#x661;` (Arabic-Indic digit one) generates the sequence ١ then ١ then ٢ ...
- 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 ....`
- A format token `w` generates numbers written as lower-case words, for example in English, `one two three four ...`
- A format token `W` generates numbers written as upper-case words, for example in English, `ONE TWO THREE FOUR ...`
- A format token `Ww` generates numbers written as title-case words, for example in English, `One Two Three Four ...`
- Any other format token indicates a numbering sequence in which that token represents the number 1 (one) (but see the note below). It is implementation-defined which numbering sequences, additional to those listed above, are supported. If an implementation does not support a numbering sequence represented by the given token, it MUST use a format token of `1`.

  Note:

  In some traditional numbering sequences additional signs are added to denote that the letters should be interpreted as numbers; these are not included in the format token. An example, see also the example below, is classical Greek where a *dexia keraia* and sometimes an *aristeri keraia* is added.

For all format tokens other than the first kind above (one that consists of decimal digits), there MAY be implementation-defined lower and upper bounds on the range of numbers that can be formatted using this format token; indeed, for some numbering sequences there may be intrinsic limits. For example, the formatting token `&#x2460;` (circled digit one) has a range of 1 to 20 imposed by the Unicode character repertoire. For the numbering sequences described above any upper bound imposed by the implementation MUST NOT be less than 1000 (one thousand) and any lower bound must not be greater than 1. Numbers that fall outside this range MUST be formatted using the format token `1`. The numbering sequence associated with the format token `1` has a lower bound of 0 (zero).

The above expansions of numbering sequences for format tokens such as `a` and `i` are indicative but not prescriptive. There are various conventions in use for how alphabetic sequences continue when the alphabet is exhausted, and differing conventions for how roman numerals are written (for example, `IV` versus `IIII` as the representation of the number 4). Sometimes alphabetic sequences are used that omit letters such as `i` and `o`. This specification does not prescribe the detail of any sequence other than those sequences consisting entirely of decimal digits.

Many numbering sequences are language-sensitive. This applies especially to the sequence selected by the tokens `w`, `W` and `Ww`. It also applies to other sequences, for example different languages using the Cyrillic alphabet use different sequences of characters, each starting with the letter #x410 (Cyrillic capital letter A). In such cases, the `lang` attribute specifies which language's conventions are to be used; it has the same range of values as `xml:lang` (see [XML 1.0]). If no `lang` value is specified, the language that is used is implementation-defined. The set of languages for which numbering is supported is implementation-defined. If a language is requested that is not supported, the processor uses the language that it would use if the `lang` attribute were omitted.

If the optional `ordinal` attribute is present, and if its value is not a zero-length string, this indicates a request to output ordinal numbers rather than cardinal numbers. For example, in English, the value `ordinal="yes"` when used with the format token `1` outputs the sequence `1st 2nd 3rd 4th ...`, and when used with the format token `w` outputs the sequence `first second third fourth ...`. In some languages, ordinal numbers vary depending on the grammatical context, for example they may have different genders and may decline with the noun that they qualify. In such cases the value of the `ordinal` attribute may be used to indicate the variation of the ordinal number required. The way in which the variation is indicated will depend on the conventions of the language. For inflected languages that vary the ending of the word, the preferred approach is to indicate the required ending, preceded by a hyphen: for example in German, appropriate values are `-e, -er, -es, -en`. It is implementation-defined what combinations of values of the format token, the language, and the `ordinal` attribute are supported. If ordinal numbering is not supported for the combination of the format token, the language, and the actual value of the `ordinal` attribute, the request is ignored and cardinal numbers are generated instead.

Example: Ordinal Numbering in Italian

The specification `format="1" ordinal="-°" lang="it"`, if supported, should produce the sequence:

```
1° 2° 3° 4° ...
```

The specification `format="Ww" ordinal="-o" lang="it"`, if supported, should produce the sequence:

```
Primo Secondo Terzo Quarto Quinto ...
```

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:

Implementations may use extension attributes on `xsl:number` to provide additional control over the way in which numbers are formatted.

The `grouping-separator` attribute gives the separator used as a grouping (for example, 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` while `grouping-separator="."` and `grouping-size="2"` would produce numbers of the form `1.00.00.00`. If only one of the `grouping-separator` and `grouping-size` attributes is specified, then it is ignored.

Example: Format Tokens and the Resulting Sequences

These examples use non-Latin characters which might not display correctly in all browsers, depending on the system configuration.

Description	Format Token	Sequence
French cardinal words	`format="Ww" lang="fr"`	Un, Deux, Trois, Quatre
German ordinal words	`format="w" ordinal="-e" lang="de"`	erste, zweite, dritte, vierte
Katakana numbering	`format="&#x30A2;"`	ア, イ, ウ, エ, オ, カ, キ, ク, ケ, コ, サ, シ, ス, セ, ソ, タ, チ, ツ, テ, ト, ナ, ニ, ヌ, ネ, ノ, ハ, ヒ, フ, ヘ, ホ, マ, ミ, ム, メ, モ, ヤ, ユ, ヨ, ラ, リ, ル, レ, ロ, ワ, ヰ, ヱ, ヲ, ン
Katakana numbering in iroha order	`format="&#x30A4;"`	イ, ロ, ハ, ニ, ホ, ヘ, ト, チ, リ, ヌ, ル, ヲ, ワ, カ, ヨ, タ, レ, ソ, ツ, ネ, ナ, ラ, ム, ウ, ヰ, ノ, オ, ク, ヤ, マ, ケ, フ, コ, エ, テ, ア, サ, キ, ユ, メ, ミ, シ, ヱ, ヒ, モ, セ, ス
Thai numbering	`format="&#x0E51;"`	๑, ๒, ๓, ๔, ๕, ๖, ๗, ๘, ๙, ๑๐, ๑๑, ๑๒, ๑๓, ๑๔, ๑๕, ๑๖, ๑๗, ๑๘, ๑๙, ๒๐
Traditional Hebrew numbering	`format="&#x05D0;" letter-value="traditional"`	א, ב, ג, ד, ה, ו, ז, ח, ט, י, יא, יב, יג, יד, טו, טז, יז, יח, יט, כ
Traditional Georgian numbering	`format="&#x10D0;" letter-value="traditional"`	ა, ბ, გ, დ, ე, ვ, ზ, ჱ, თ, ი, ია, იბ, იგ, იდ, იე, ივ, იზ, იჱ, ით, კ
Classical Greek numbering (see note)	`format="&#x03B1;" letter-value="traditional"`	α′, β′, γ′, δ′, ε′, ϛ′, ζ′, η′, θ′, ι′, ια′, ιβ′, ιγ′, ιδ′, ιε′, ιϛ′, ιζ′, ιη′, ιθ′, κ′
Old Slavic numbering	`format="&#x0430;" letter-value="traditional"`	A, B, Г, Д, E, S, З, И, Ѳ, I, AI, BI, ГI, ДI, EI, SI, ЗI, ИI, ѲI, K

Note that Glassical Greek is an example where the format token is not the same as the representation of the number 1.

13 Sorting

[DEFINITION: A **sort key specification** is a sequence of one or more adjacent `xsl:sort` elements which together define rules for sorting the items in an input sequence to form a sorted sequence.]

[DEFINITION: Within a sort key specification, each `xsl:sort` element defines one **sort key component**.] The first `xsl:sort` element specifies the primary component of the sort key specification, the second `xsl:sort` element specifies the secondary component of the sort key specification and so on.

A sort key specification may occur immediately within an `xsl:apply-templates`, `xsl:for-each`, `xsl:perform-sort`, or `xsl:for-each-group` element.

Note:

When used within `xsl:for-each`, `xsl:for-each-group`, or `xsl:perform-sort`, `xsl:sort` elements must occur before any other children.

13.1 The `xsl:sort` Element

```
<xsl:sort
  select? = expression
  lang? = { nmtoken }
  order? = { "ascending" | "descending" }
  collation? = { uri }
  stable? = { "yes" | "no" }
  case-order? = { "upper-first" | "lower-first" }
  data-type? = { "text" | "number" | qname-but-not-ncname }>
  <!-- Content: sequence-constructor -->
</xsl:sort>
```

The `xsl:sort` element defines a sort key component. A sort key component specifies how a sort key value is to be computed for each item in the sequence being sorted, and also how two sort key values are to be compared.

The value of a sort key component is determined either by its `select` attribute, or by the contained sequence constructor. If neither is present, the default is `select="."`, which has the effect of sorting on the actual value of the item if it is an atomic value, or on the typed-value of the item if it is a node. If a `select` attribute is present, its value MUST be an XPath expression.

[ERR XTSE1015] It is a static error if an `xsl:sort` element with a `select` attribute has non-empty content.

Those attributes of the `xsl:sort` elements whose values are attribute value templates are evaluated using the same focus as is used to evaluate the `select` attribute of the containing instruction (specifically, `xsl:apply-templates`, `xsl:for-each`, `xsl:for-each-group`, or `xsl:perform-sort`).

The `stable` attribute is permitted only on the first `xsl:sort` element within a sort key specification

[ERR XTSE1017] It is a static error if an `xsl:sort` element other than the first in a sequence of sibling `xsl:sort` elements has a `stable` attribute.

[DEFINITION: A sort key specification is said to be **stable** if its first `xsl:sort` element has no `stable` attribute, or has a `stable` attribute whose effective value is `yes`.]

13.1.1 The Sorting Process

[DEFINITION: The sequence to be sorted is referred to as the **initial sequence**.]

[DEFINITION: The sequence after sorting as defined by the `xsl:sort` elements is referred to as the **sorted sequence**.]

[DEFINITION: For each item in the initial sequence, a value is computed for each sort key component within the sort key specification. The value computed for an item by using the *N*th sort key component is referred to as the *N*th **sort key value** of that item.]

The items in the initial sequence are ordered into a sorted sequence by comparing their sort key values. The relative position of two items *A* and *B* in the sorted sequence is determined as follows. The first sort key value of *A* is compared with the first sort key value of *B*, according to the rules of the first sort key component. If, under these rules, *A* is less than *B*, then *A* will precede *B* in the sorted sequence, unless the `order` attribute of this sort key component specifies `descending`, in which case *B* will precede *A* in the sorted sequence. If, however, the relevant sort key values compare equal, then the second sort key value of *A* is compared with the second sort key value of *B*, according to the rules of the second sort key component. This continues until two sort key values are found that compare unequal. If all the sort key values compare equal, and the sort key specification is stable, then *A* will precede *B* in the sorted sequence if and only if *A* preceded *B* in the initial sequence. If all the sort key values compare equal, and the sort key specification is not stable, then the relative order of *A* and *B* in the sorted sequence is implementation-dependent.

> **Note:**
>
> If two items have equal sort key values, and the sort is stable, then their order in the sorted sequence will be the same as their order in the initial sequence, regardless of whether `order="descending"` was specified on any or all of the sort key components.

The *N*th sort key value is computed by evaluating either the `select` attribute or the contained sequence constructor of the *N*th `xsl:sort` element, or the expression `.` (dot) if neither is present. This evaluation is done with the focus set as follows:

- The context item is the item in the initial sequence whose sort key value is being computed.
- The context position is the position of that item in the initial sequence.
- The context size is the size of the initial sequence.

> **Note:**
>
> As in any other XPath expression, the `current` function may be used within the `select` expression of `xsl:sort` to refer to the item that is the context item for the expression as a whole; that is, the item whose sort key value is being computed.

The sort key values are atomized, and are then compared. The way they are compared depends on their data type, as described in the next section.

13.1.2 Comparing Sort Key Values

It is possible to force the system to compare sort key values using the rules for a particular data type by including a cast as part of the sort key component. For example, `<xsl:sort select="xs:date(@dob)"/>` will force the attributes to be compared as dates. In the absence of such a cast, the sort key values are compared using the rules appropriate to their data type. Any values of type `xs:untypedAtomic` are cast to `xs:string`.

For backwards compatibility with XSLT 1.0, the `data-type` attribute remains available. If this has the effective value `text`, the atomized sort key values are converted to strings before being compared. If it has the effective value `number`, the atomized sort key values are converted to doubles before being compared. The conversion is done by using the `string`^{FO} or `number`^{FO} function as appropriate. If the `data-type` attribute has any other effective value, then the value MUST be a lexical QName with a non-empty prefix, and the effect of the attribute is implementation-defined.

[ERR XTTE1020] If any sort key value, after atomization and any type conversion REQUIRED by the `data-type` attribute, is a sequence containing more than one item, then the effect depends on whether the `xsl:sort` element is evaluated with backwards compatible behavior. With backwards compatible behavior, the effective sort key value is the first item in the sequence. In other cases, this is a type error.

The set of sort key values (after any conversion) is first divided into two categories: empty values, and ordinary values. The empty sort key values represent those items where the sort key value is an empty sequence. These values are considered for sorting purposes to be equal to each other, but less than any other value. The remaining values are classified as ordinary values.

[ERR XTDE1030] It is a non-recoverable dynamic error if, for any sort key component, the set of sort key values evaluated for all the items in the initial sequence, after any type conversion requested, contains a pair of ordinary values for which the result of the XPath `lt` operator is an error.

Note:

The above error condition may occur if the values to be sorted are of a type that does not support ordering (for example, `xs:QName`) or if the sequence is heterogeneous (for example, if it contains both strings and numbers). The error can generally be prevented by invoking a cast or constructor function within the sort key component.

The error condition is subject to the usual caveat that a processor is not required to evaluate any expression solely in order to determine whether it raises an error. For example, if there are several sort key components, then a processor is not required to evaluate or compare minor sort key values unless the corresponding major sort key values are equal.

In general, comparison of two ordinary values is performed according to the rules of the XPath `lt` operator. To ensure a total ordering, the same implementation of the `lt` operator MUST be used for all the comparisons: the one that is chosen is the one appropriate to the most specific type to which all the values can be converted by subtype substitution and/or type promotion. For example, if the sequence contains both `xs:decimal` and `xs:double` values, then the values are compared using `xs:double` comparison, even when comparing two `xs:decimal` values. NaN values, for sorting purposes, are considered to be equal to each other, and less than any other numeric value. Special rules also apply to the `xs:string` and `xs:anyURI` types, and types derived by restriction therefrom,, as described in the next section.

13.1.3 Sorting Using Collations

The rules given in this section apply when comparing values whose type is `xs:string` or a type derived by restriction from `xs:string`, or whose type is `xs:anyURI` or a type derived by restriction from `xs:anyURI`.

[DEFINITION: Facilities in XSLT 2.0 and XPath 2.0 that require strings to be ordered rely on the concept of a named **collation**. A collation is a set of rules that determine whether two strings are equal, and if not, which of them is to be sorted before the other.] A collation is identified by a URI, but the manner in which this URI is associated with an actual rule or algorithm is implementation-defined.

The one collation URI that must be recognized by every implementation is `http://www.w3.org/2005/xpath-functions/collation/codepoint`, which provides the ability to compare strings based on the Unicode codepoint values of the characters in the string.

For more information about collations, see Section 7.3 Equality and Comparison of Strings[FO] in [Functions and Operators]. Some specifications, for example [UNICODE TR10], use the term "collation" to describe rules that can be tailored or parameterized for various purposes. In this specification, a collation URI refers to a collation in which all such parameters have already been fixed. Therefore, if a collation URI is specified, other attributes such as `case-order` and `lang` are ignored.

Note:

The reason XSLT does not provide detailed mechanisms for defining collating sequences is that many implementations will re-use collating mechanisms available from the underlying implementation platform (for example, from the operating system or from the run-time library of a chosen programming language). These will inevitably differ from one XSLT implementation to another.

If the `xsl:sort` element has a `collation` attribute, then the strings are compared according to the rules for the named collation: that is, they are compared using the XPath function call `compare($a, $b, $collation)`.

If the effective value of the `collation` attribute of `xsl:sort` is a relative URI, then it is resolved against the base URI of the `xsl:sort` element.

[ERR XTDE1035] It is a non-recoverable dynamic error if the `collation` attribute of `xsl:sort` (after resolving against the base URI) is not a URI that is recognized by the implementation as referring to a collation.

Note:

It is entirely for the implementation to determine whether it recognizes a particular collation URI. For example, if the implementation allows collation URIs to contain parameters in the query part of the URI, it is the implementation that determines whether a URI containing an unknown or invalid parameter is or is not a recognized collation URI. The fact that this error is described as non-recoverable thus does not prevent an implementation applying a fallback collation if it chooses to do so.

The `lang` and `case-order` attributes are ignored if a `collation` attribute is present. But in the absence of a `collation` attribute, these attributes provide input to an implementation-defined algorithm to locate a suitable collation:

- The `lang` attribute indicates that a collation suitable for a particular natural language SHOULD be used. The effective value of the attribute MUST be a value that would be valid for the `xml:lang` attribute (see [XML 1.0]).
- The `case-order` attribute indicates whether the desired collation SHOULD sort upper-case letters before lower-case or vice versa. The effective value of the attribute MUST be either `lower-first` (indicating that lower-case letters precede upper-case letters in the collating sequence) or `upper-first` (indicating that upper-case letters precede lower-case).

If none of the `collation`, `lang` or `case-order` attributes is present, the collation is chosen in an implementation-defined way. It is not REQUIRED that the default collation for sorting should be the same as the default collation used when evaluating XPath expressions, as described in *5.4.1 Initializing the Static Context* and *3.6.1 The default-collation attribute*.

Note:

It is usually appropriate, when sorting, to use a strong collation, that is, one that takes account of secondary differences (accents) and tertiary differences (case) between strings that are otherwise equal. A weak collation, which ignores such differences, may be more suitable when comparing strings for equality.

Useful background information on international sorting is provided in [UNICODE TR10]. The `case-order` attribute may be interpreted as described in section 6.6 of [UNICODE TR10].

13.2 Creating a Sorted Sequence

```
<!-- Category: instruction -->
<xsl:perform-sort
  select? = expression>
  <!-- Content: (xsl:sort+, sequence-constructor) -->
</xsl:perform-sort>
```

The `xsl:perform-sort` instruction is used to return a sorted sequence.

The initial sequence is obtained either by evaluating the `select` attribute or by evaluating the contained sequence constructor (but not both). If there is no `select` attribute and no sequence constructor then the initial sequence (and therefore, the sorted sequence) is an empty sequence.

[ERR XTSE1040] It is a static error if an `xsl:perform-sort` instruction with a `select` attribute has any content other than `xsl:sort` and `xsl:fallback` instructions.

The result of the `xsl:perform-sort` instruction is the result of sorting its initial sequence using its contained sort key specification.

Example: Sorting a Sequence of Atomic Values

The following stylesheet function sorts a sequence of atomic values using the value itself as the sort key.

```
<xsl:function name="local:sort" as="xs:anyAtomicType*">
  <xsl:param name="in" as="xs:anyAtomicType*"/>
  <xsl:perform-sort select="$in">
    <xsl:sort select="."/>
  </xsl:perform-sort>
</xsl:function>
```

Example: Writing a Function to Perform a Sort

The following example defines a function that sorts books by price, and uses this function to output the five books that have the lowest prices:

```
<xsl:function name="bib:books-by-price" as="schema-element(bib:book)*">
  <xsl:param name="in" as="schema-element(bib:book)*"/>
  <xsl:perform-sort select="$in">
    <xsl:sort select="xs:decimal(bib:price)"/>
  </xsl:perform-sort>
</xsl:function>
   ...
   <xsl:copy-of select="bib:books-by-price(//bib:book)[position() = 1 to 5]"/>
```

13.3 Processing a Sequence in Sorted Order

When used within `xsl:for-each` or `xsl:apply-templates`, a sort key specification indicates that the sequence of items selected by that instruction is to be processed in sorted order, not in the order of the supplied sequence.

Example: Processing Elements in Sorted 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>
```

When used within `xsl:for-each-group`, a sort key specification indicates the order in which the groups are to be processed. For the effect of `xsl:for-each-group`, see *14 Grouping*.

14 Grouping

The facilities described in this section are designed to allow items in a sequence to be grouped based on common values; for example it allows grouping of elements having the same value for a particular attribute, or elements with the same name, or elements with common values for any other expression. Since grouping identifies items with duplicate values, the same facilities also allow selection of the distinct values in a sequence of items, that is, the elimination of duplicates.

> **Note:**
>
> Simple elimination of duplicates can also be achieved using the function `distinct-values`[FO] in the core function library: see [Functions and Operators].

In addition these facilities allow grouping based on sequential position, for example selecting groups of adjacent `para` elements. The facilities also provide an easy way to do fixed-size grouping, for example identifying groups of three adjacent nodes, which is useful when arranging data in multiple columns.

For each group of items identified, it is possible to evaluate a sequence constructor for the group. Grouping is nestable to multiple levels so that groups of distinct items can be identified, then from among the distinct groups selected, further sub-grouping of distinct items in the current group can be done.

It is also possible for one item to participate in more than one group.

14.1 The Current Group

```
current-group() as item()*
```

[DEFINITION: The evaluation context for XPath expressions includes a component called the **current group**, which is a sequence. The current group is the collection of related items that are processed collectively in one iteration of the `xsl:for-each-group` element.]

While an `xsl:for-each-group` instruction is being evaluated, the current group will be non-empty. At other times, it will be an empty sequence.

The function `current-group` returns the current group.

The function takes no arguments.

[ERR XTSE1060] It is a static error if the `current-group` function is used within a pattern.

14.2 The Current Grouping Key

```
current-grouping-key() as xs:anyAtomicType?
```

[DEFINITION: The evaluation context for XPath expressions includes a component called the **current grouping key**, which is an atomic value. The current grouping key is the grouping key shared in common by all the items within the current group.]

While an `xsl:for-each-group` instruction with a `group-by` or `group-adjacent` attribute is being evaluated, the current grouping key will be a single atomic value. At other times, it will be the empty sequence.

The function `current-grouping-key` returns the current grouping key.

Although the grouping keys of all items in a group are by definition equal, they are not necessarily identical. For example, one might be an `xs:float` while another is an `xs:decimal`. The `current-grouping-key` function is defined to return the grouping key of the initial item in the group, after atomization and casting of `xs:untypedAtomic` to `xs:string`.

The function takes no arguments.

[ERR XTSE1070] It is a static error if the `current-grouping-key` function is used within a pattern.

14.3 The `xsl:for-each-group` Element

```
<!-- Category: instruction -->
<xsl:for-each-group
  select = expression
  group-by? = expression
  group-adjacent? = expression
  group-starting-with? = pattern
  group-ending-with? = pattern
  collation? = { uri }>
  <!-- Content: (xsl:sort*, sequence-constructor) -->
</xsl:for-each-group>
```

This element is an instruction that may be used anywhere within a sequence constructor.

[DEFINITION: The `xsl:for-each-group` instruction allocates the items in an input sequence into **groups** of items (that is, it establishes a collection of sequences) based either on common values of a grouping key, or on a pattern that the initial or final node in a group must match.] The sequence constructor that forms the content of the `xsl:for-each-group` instruction is evaluated once for each of these groups.

[DEFINITION: The sequence of items to be grouped, which is referred to as the **population**, is determined by evaluating the XPath expression contained in the `select` attribute.]

[DEFINITION: The population is treated as a sequence; the order of items in this sequence is referred to as **population order**].

A group is never empty. If the population is empty, the number of groups will be zero. The assignment of items to groups depends on the `group-by`, `group-adjacent`, `group-starting-with`, and `group-ending-with` attributes.

[ERR XTSE1080] These four attributes are mutually exclusive: it is a static error if none of these four attributes is present, or if more than one of them is present.

[ERR XTSE1090] It is an error to specify the `collation` attribute if neither the `group-by` attribute nor `group-adjacent` attribute is specified.

[DEFINITION: If either of the `group-by` attribute or `group-adjacent` attributes is present, then **grouping keys** are calculated for each item in the population. The grouping keys are the items in the sequence obtained by evaluating the expression contained in the `group-by` attribute or `group-adjacent` attribute, atomizing the result, and then casting an `xs:untypedAtomic` value to `xs:string`.]

When calculating grouping keys for an item in the population, the expression contained in the `group-by` or `group-adjacent` attribute is evaluated with that item as the context item, with its position in population order as the context position, and with the size of the population as the context size. The resulting sequence is atomized, and each atomic value in the atomized sequence acts as a grouping key for that item in the population.

If the `group-by` attribute is present, then an item in the population may have multiple grouping keys: that is, the `group-by` expression evaluates to a sequence. The item is included in as many groups as there are distinct grouping keys (which may be zero). If the `group-adjacent` attribute is used, then each item in the population MUST have exactly one grouping key value.

[ERR XTTE1100] It is a type error if the grouping key evaluated using the `group-adjacent` attribute is an empty sequence, or a sequence containing more than one item.

Grouping keys are compared using the rules for the `eq` operator appropriate to their dynamic type. Values of type `xs:untypedAtomic` are cast to `xs:string` before the comparison. Two items that are not comparable using the `eq` operator are considered to be not equal, that is, they are allocated to different groups. If the values are strings, or untyped atomic values, then if there is a `collation` attribute the values are compared using the collation specified as the effective value of the `collation` attribute, resolved if relative against the base URI of the `xsl:for-each-group` element. If there is no `collation` attribute then the default collation is used.

For the purposes of grouping, the value `NaN` is considered equal to itself.

[ERR XTDE1110] It is a non-recoverable dynamic error if the collation URI specified to `xsl:for-each-group` (after resolving against the base URI) is a collation that is not recognized by the implementation. (For notes, [see ERR XTDE1035].)

For more information on collations, see *13.1.3 Sorting Using Collations*.

[ERR XTTE1120] When the `group-starting-with` or `group-ending-with` attribute is used, it is a type error if the result of evaluating the `select` expression contains an item that is not a node.

- If the `group-by` attribute is present, the items in the population are examined, in population order. For each item *J*, the expression in the `group-by` attribute is evaluated to produce a sequence of zero or more grouping key values. For each one of these grouping keys, if there is already a group created to hold items having that grouping key value, *J* is added to that group; otherwise a new group is created for items with that grouping key value, and *J* becomes its first member.

  An item in the population may thus be assigned to zero, one, or many groups. An item will never be assigned more than once to the same group; if two or more grouping keys for the same item are equal, then the duplicates are ignored. An *item* here means the item at a particular position within the population—if the population contains the same node at several different positions in the sequence then a group may indeed contain duplicate nodes.

  The number of groups will be the same as the number of distinct grouping key values present in the population.
- If the `group-adjacent` attribute is present, the items in the population are examined, in population order. If an item has the same value for the grouping key as its preceding item within the population (in population order), then it is assigned to the same group as its preceding item; otherwise a new group is created and the item becomes its first member.
- If the `group-starting-with` attribute is present, then its value MUST be a pattern. In this case, the items in the population MUST all be nodes.

  The nodes in the population are examined in population order. If a node matches the pattern, or is the first node in the population, then a new group is created and the node becomes its first member. Otherwise, the node is assigned to the same group as its preceding node within the population.
- If the `group-ending-with` attribute is present, then its value MUST be a pattern. In this case, the items in the population MUST all be nodes.

  The nodes in the population are examined in population order. If a node is the first node in the population, or if the previous node in the population matches the pattern, then a new group is created and the node becomes its first member. Otherwise, the node is assigned to the same group as its preceding node within the population.

[DEFINITION: For each group, the item within the group that is first in population order is known as the **initial item** of the group.]

[DEFINITION: There is an ordering among groups referred to as the **order of first appearance**. A group *G* is defined to precede a group *H* in order of first appearance if the initial item of *G* precedes the initial item of *H* in population order. If two groups *G* and *H* have the same initial item (because the item is in both groups) then *G* precedes *H* if the grouping key of *G* precedes the grouping key of *H* in the sequence that results from evaluating the `group-by` expression of this initial item.]

[DEFINITION: There is another ordering among groups referred to as **processing order**. If group *R* precedes group *S* in processing order, then in the result sequence returned by the `xsl:for-each-group` instruction the items generated by processing group *R* will precede the items generated by processing group *S*.]

If there are no `xsl:sort` elements immediately within the `xsl:for-each-group` element, the processing order of the groups is the order of first appearance.

Otherwise, the `xsl:sort` elements immediately within the `xsl:for-each-group` element define the processing order of the groups (see *13 Sorting*). They do not affect the order of items within each group. Multiple sort key components are allowed, and are evaluated in major-to-minor order. If two groups have the same values for all their sort key components, they are processed in order of first appearance.

The `select` expression of an `xsl:sort` element is evaluated once for each group. During this evaluation, the context item is the initial item of the group, the context position is the position of this item within the set of initial items (that is, one item for each group in the population) in population order, the context size is the number of groups, the current group is the group whose sort key value is being determined, and the current grouping key is the grouping key for that group. If the `xsl:for-each-group` instruction uses the `group-starting-with` or `group-ending-with` attributes, then the current grouping key is the empty sequence.

Example: Sorting Groups

For example, this means that if the grouping key is `@category`, you can sort the groups in order of their grouping key by writing `<xsl:sort select="current-grouping-key()"/>`; or you can sort the groups in order of size by writing `<xsl:sort select="count (current-group())"/>`

The sequence constructor contained in the `xsl:for-each-group` element is evaluated once for each of the groups, in processing order. The sequences that result are concatenated, in processing order, to form the result of the `xsl:for-each-group` element. Within the sequence constructor, the context item is the initial item of the relevant group, the context position is the position of this item among the sequence of initial items (one item for each group) arranged in processing order of the groups, the context size is the number of groups, the current group is the group being processed, and the current grouping key is the grouping key for that group. If the `xsl:for-each-group` instruction uses the `group-starting-with` or `group-ending-with` attributes, then the current grouping key is the empty sequence. This has the effect that within the sequence constructor, a call on `position()` takes successive values `1, 2, ... last()`.

During the evaluation of a stylesheet function, the current group and current grouping key are set to the empty sequence, and revert to their previous values on completion of evaluation of the stylesheet function.

On completion of the evaluation of the `xsl:for-each-group` instruction, the current group and current grouping key revert to their previous value.

14.4 Examples of Grouping

Example: Grouping Nodes based on Common Values

The following example groups a list of nodes based on common values. The resulting groups are numbered but unsorted, and a total is calculated for each group.

Source XML document:

```
<cities>
  <city name="Milano"  country="Italia"      pop="5"/>
  <city name="Paris"   country="France"      pop="7"/>
  <city name="München" country="Deutschland" pop="4"/>
  <city name="Lyon"    country="France"      pop="2"/>
  <city name="Venezia" country="Italia"      pop="1"/>
</cities>
```

More specifically, the aim is to produce a four-column table, containing one row for each distinct country. The four columns are to contain first, a sequence number giving the number of the row; second, the name of the country, third, a comma-separated alphabetical list of the city names within that country, and fourth, the sum of the `pop` attribute for the cities in that country.

Desired output:

```
<table>
  <tr>
    <th>Position</th>
    <th>Country</th>
    <th>List of Cities</th>
    <th>Population</th>
  </tr>
  <tr>
    <td>1</td>
    <td>Italia</td>
    <td>Milano, Venezia</td>
    <td>6</td>
  </tr>
  <tr>
    <td>2</td>
    <td>France</td>
    <td>Lyon, Paris</td>
    <td>9</td>
  </tr>
  <tr>
    <td>3</td>
    <td>Deutschland</td>
    <td>München</td>
    <td>4</td>
  </tr>
</table>
```

Solution:

```
<table xsl:version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <tr>
    <th>Position</th>
    <th>Country</th>
    <th>City List</th>
    <th>Population</th>
  </tr>
  <xsl:for-each-group select="cities/city" group-by="@country">
    <tr>
      <td><xsl:value-of select="position()"/></td>
      <td><xsl:value-of select="@country"/></td>
      <td>
        <xsl:value-of select="current-group()/@name" separator=", "/>
      </td>
      <td><xsl:value-of select="sum(current-group()/@pop)"/></td>
```

```
    </tr>
  </xsl:for-each-group>
</table>
```

Example: A Composite Grouping Key

Sometimes it is necessary to use a composite grouping key: for example, suppose the source document is similar to the one used in the previous examples, but allows multiple entries for the same country and city, such as:

```
<cities>
  <city name="Milano"  country="Italia"  year="1950"   pop="5.23"/>
  <city name="Milano"  country="Italia"  year="1960"   pop="5.29"/>
  <city name="Padova"  country="Italia"  year="1950"   pop="0.69"/>
  <city name="Padova"  country="Italia"  year="1960"   pop="0.93"/>
  <city name="Paris"   country="France"  year="1951"   pop="7.2"/>
  <city name="Paris"   country="France"  year="1961"   pop="7.6"/>
</cities>
```

Now suppose we want to list the average value of `@pop` for each (country, name) combination. One way to handle this is to concatenate the parts of the key, for example `<xsl:for-each-group select="concat(@country, '/', @name)">`. A more flexible solution is to nest one `xsl:for-each-group` element directly inside another:

```
<xsl:for-each-group select="cities/city" group-by="@country">
  <xsl:for-each-group select="current-group()" group-by="@name">
    <p><xsl:value-of select="@name"/>, <xsl:value-of select="@country"/>:
        <xsl:value-of select="avg(current-group()/@pop)"/></p>
  </xsl:for-each-group>
</xsl:for-each-group>
```

The two approaches are not precisely equivalent. If the code were changed to output the value of `position()` alongside `@name` then the first approach (a single `xsl:for-each-group` element with a compound key) would number the groups (1, 2, 3), while the second approach (two nested `xsl:for-each-group` elements) would number them (1, 2, 1).

Example: Identifying a Group by its Initial Element

The next example identifies a group not by the presence of a common value, but rather by adjacency in document order. A group consists of an `h2` element, followed by all the `p` elements up to the next `h2` element.

Source XML document:

```
<body>
  <h2>Introduction</h2>
  <p>XSLT is used to write stylesheets.</p>
  <p>XQuery is used to query XML databases.</p>
  <h2>What is a stylesheet?</h2>
  <p>A stylesheet is an XML document used to define a transformation.</p>
  <p>Stylesheets may be written in XSLT.</p>
  <p>XSLT 2.0 introduces new grouping constructs.</p>
</body>
```

Desired output:

```
<chapter>
  <section title="Introduction">
    <para>XSLT is used to write stylesheets.</para>
    <para>XQuery is used to query XML databases.</para>
  </section>
  <section title="What is a stylesheet?">
    <para>A stylesheet is an XML document used to define a transformation.</para>
    <para>Stylesheets may be written in XSLT.</para>
    <para>XSLT 2.0 introduces new grouping constructs.</para>
  </section>
</chapter>
```

Solution:

```
<xsl:template match="body">
  <chapter>
        <xsl:for-each-group select="*" group-starting-with="h2"      >
          <section title="{self::h2}">
            <xsl:for-each select="current-group()[self::p]">
              <para><xsl:value-of select="."/></para>
            </xsl:for-each>
          </section>
        </xsl:for-each-group>
  </chapter>
</xsl:template>
```

The use of `title="{self::h2}"` rather than `title="{.}"` is to handle the case where the first element is not an `h2` element.

Example: Identifying a Group by its Final Element

The next example illustrates how a group of related elements can be identified by the last element in the group, rather than the first. Here the absence of the attribute `continued="yes"` indicates the end of the group.

Source XML document:

```
<doc>
  <page continued="yes">Some text</page>
  <page continued="yes">More text</page>
  <page>Yet more text</page>
  <page continued="yes">Some words</page>
  <page continued="yes">More words</page>
  <page>Yet more words</page>
</doc>
```

Desired output:

```
<doc>
  <pageset>
    <page>Some text</page>
    <page>More text</page>
    <page>Yet more text</page>
  </pageset>
  <pageset>
    <page>Some words</page>
    <page>More words</page>
    <page>Yet more words</page>
  </pageset>
</doc>
```

Solution:

```
<xsl:template match="doc">
<doc>
  <xsl:for-each-group select="*"
                      group-ending-with="page[not(@continued='yes')]">
    <pageset>
      <xsl:for-each select="current-group()">
        <page><xsl:value-of select="."/></page>
      </xsl:for-each>
    </pageset>
  </xsl:for-each-group>
</doc>
</xsl:template>
```

Example: Adding an Element to Several Groups

The next example shows how an item can be added to multiple groups. Book titles will be added to one group for each indexing term marked up within the title.

Source XML document:

```
<titles>
    <title>A Beginner's Guide to <ix>Java</ix></title>
    <title>Learning <ix>XML</ix></title>
    <title>Using <ix>XML</ix> with <ix>Java</ix></title>
</titles>
```

Desired output:

```
<h2>Java</h2>
    <p>A Beginner's Guide to Java</p>
    <p>Using XML with Java</p>
<h2>XML</h2>
    <p>Learning XML</p>
    <p>Using XML with Java</p>
```

Solution:

```
<xsl:template match="titles">
    <xsl:for-each-group select="title" group-by="ix">
      <h2><xsl:value-of select="current-grouping-key()"/></h2>
      <xsl:for-each select="current-group()">
        <p><xsl:value-of select="."/></p>
      </xsl:for-each>
    </xsl:for-each-group>
</xsl:template>
```

Example: Grouping Alternating Sequences of Elements

In the final example, the membership of a node within a group is based both on adjacency of the nodes in document order, and on common values. In this case, the grouping key is a boolean condition, true or false, so the effect is that a grouping establishes a maximal sequence of nodes for which the condition is true, followed by a maximal sequence for which it is false, and so on.

Source XML document:

```
<p>Do <em>not</em>:
    <ul>
    <li>talk,</li>
    <li>eat, or</li>
    <li>use your mobile telephone</li>
    </ul>
    while you are in the cinema.</p>
```

Desired output:

```
<p>Do <em>not</em>:</p>
    <ul>
    <li>talk,</li>
    <li>eat, or</li>
    <li>use your mobile telephone</li>
    </ul>
    <p>while you are in the cinema.</p>
```

Solution:

This requires creating a `p` element around the maximal sequence of sibling nodes that does not include a `ul` or `ol` element.

This can be done by using `group-adjacent`, with a grouping key that is true if the element is a `ul` or `ol` element, and false otherwise:

```
<xsl:template match="p">
    <xsl:for-each-group select="node()"
            group-adjacent="self::ul or self::ol">
        <xsl:choose>
            <xsl:when test="current-grouping-key()">
                <xsl:copy-of select="current-group()"/>
            </xsl:when>
            <xsl:otherwise>
                <p>
                    <xsl:copy-of select="current-group()"/>
                </p>
            </xsl:otherwise>
        </xsl:choose>
    </xsl:for-each-group>
</xsl:template>
```

15 Regular Expressions

The core function library for XPath 2.0 defines three functions that make use of regular expressions:

- matches[FO] returns a boolean result that indicates whether or not a string matches a given regular expression.
- replace[FO] takes a string as input and returns a string obtained by replacing all substrings that match a given regular expression with a replacement string.
- tokenize[FO] returns a sequence of strings formed by breaking a supplied input string at any separator that matches a given regular expression.

These functions are described in [Functions and Operators].

For more complex string processing than is possible using these functions, XSLT provides an instruction `xsl:analyze-string`, which is defined in this section.

The regular expressions used by this instruction, and the flags that control the interpretation of these regular expressions, MUST conform to the syntax defined in [Functions and Operators] (see Section 7.6.1 Regular Expression Syntax[FO]), which is itself based on the syntax defined in [XML Schema Part 2].

15.1 The `xsl:analyze-string` instruction

```
<!-- Category: instruction -->
<xsl:analyze-string
  select = expression
  regex = { string }
  flags? = { string }>
  <!-- Content: (xsl:matching-substring?, xsl:non-matching-substring?, xsl:fallback*) -->
</xsl:analyze-string>
```

```
<xsl:matching-substring>
  <!-- Content: sequence-constructor -->
</xsl:matching-substring>
```

```
<xsl:non-matching-substring>
```

```
  <!-- Content: sequence-constructor -->
</xsl:non-matching-substring>
```

The xsl:analyze-string instruction takes as input a string (the result of evaluating the expression in the select attribute) and a regular expression (the effective value of the regex attribute).

If the result of evaluating the select expression is not a string, it is converted to a string by applying the function conversion rules.

The flags attribute may be used to control the interpretation of the regular expression. If the attribute is omitted, the effect is the same as supplying a zero-length string. This is interpreted in the same way as the $flags attribute of the functions matches FO, replace FO, and tokenize FO. Specifically, if it contains the letter m, the match operates in multiline mode. If it contains the letter s, it operates in dot-all mode. If it contains the letter i, it operates in case-insensitive mode. If it contains the letter x, then whitespace within the regular expression is ignored. For more detailed specifications of these modes, see [Functions and Operators] (Section 7.6.1.1 Flags FO).

Note:

Because the regex attribute is an attribute value template, curly brackets within the regular expression must be doubled. For example, to match a sequence of one to five characters, write regex=".{{1,5}}". For regular expressions containing many curly brackets it may be more convenient to use a notation such as regex="{'[0-9]{1,5}[a-z]{3}[0-9]{1,2}'}", or to use a variable.

The content of the xsl:analyze-string instruction must take one of the following forms:

1. A single xsl:matching-substring instruction, followed by zero or more xsl:fallback instructions
2. A single xsl:non-matching-substring instruction, followed by zero or more xsl:fallback instructions
3. A single xsl:matching-substring instruction, followed by a single xsl:non-matching-substring instruction, followed by zero or more xsl:fallback instructions

[ERR XTSE1130] It is a static error if the xsl:analyze-string instruction contains neither an xsl:matching-substring nor an xsl:non-matching-substring element.

Any xsl:fallback elements among the children of the xsl:analyze-string instruction are ignored by an XSLT 2.0 processor, but allow fallback behavior to be defined when the stylesheet is used with an XSLT 1.0 processor operating in forwards-compatible mode.

This instruction is designed to process all the non-overlapping substrings of the input string that match the regular expression supplied.

[ERR XTDE1140] It is a non-recoverable dynamic error if the effective value of the regex attribute does not conform to the REQUIRED syntax for regular expressions, as specified in [Functions and Operators]. If the regular expression is known statically (for example, if the attribute does not contain any expressions enclosed in curly brackets) then the processor MAY signal the error as a static error.

[ERR XTDE1145] It is a non-recoverable dynamic error if the effective value of the flags attribute has a value other than the values defined in [Functions and Operators]. If the value of the attribute is known statically (for example, if the attribute does not contain any expressions enclosed in curly brackets) then the processor MAY signal the error as a static error.

[ERR XTDE1150] It is a non-recoverable dynamic error if the effective value of the regex attribute is a regular expression that matches a zero-length string: or more specifically, if the regular expression $r and flags $f are such that matches("", $r, $f) returns true. If the regular expression is known statically (for example, if the attribute does not contain any expressions enclosed in curly brackets) then the processor MAY signal the error as a static error.

The xsl:analyze-string instruction starts at the beginning of the input string and attempts to find the first substring that matches the regular expression. If there are several matches, the first match is defined to be the one whose starting position comes first in the string. If several alternatives within the regular expression both match at the same position in the input string, then the match that is chosen is the first alternative that matches. For example, if the input string is The quick brown fox jumps and the regular expression is jump|jumps, then the match that is chosen is jump.

Having found the first match, the instruction proceeds to find the second and subsequent matches by repeating the search, starting at the first character that was not included in the previous match.

The input string is thus partitioned into a sequence of substrings, some of which match the regular expression, others which do not match it. Each substring will contain at least one character. This sequence of substrings is processed using the xsl:matching-substring and xsl:non-matching-substring child instructions. A matching substring is processed using the xsl:matching-substring element, a non-matching substring using the xsl:non-matching-substring element. Each of these elements takes a sequence constructor as its content. If the element is absent, the effect is the same as if it were present with empty content. In processing each substring, the contents of the substring will be the context item (as a value of type xs:string); the position of the substring within the sequence of matching and non-matching substrings will be the context position; and the number of matching and non-matching substrings will be the context size.

If the input is a zero-length string, the number of substrings will be zero, so neither the xsl:matching-substring nor xsl:non-matching-substring elements will be evaluated.

15.2 Captured Substrings

```
regex-group($group-number as xs:integer) as xs:string
```

[DEFINITION: While the xsl:matching-substring instruction is active, a set of **current captured substrings** is available, corresponding to the parenthesized sub-expressions of the regular expression.] These captured substrings are accessible using the function regex-group. This function takes an integer argument to identify the group, and returns a string representing the captured substring.

The *N*th captured substring (where *N* > 0) is the string matched by the subexpression contained by the *N*th left parenthesis in the regex. The zeroeth captured substring is the string that matches the entire regex. This means that the value of regex-group(0) is initially the same as the value of . (dot).

The function returns the zero-length string if there is no captured substring with the relevant number. This can occur for a number of reasons:

1. The number is negative.

2. The regular expression does not contain a parenthesized sub-expression with the given number.
3. The parenthesized sub-expression exists, and did not match any part of the input string.
4. The parenthesized sub-expression exists, and matched a zero-length substring of the input string.

The set of captured substrings is a context variable with dynamic scope. It is initially an empty sequence. During the evaluation of an `xsl:matching-substring` instruction it is set to the sequence of matched substrings for that regex match. During the evaluation of an `xsl:non-matching-substring` instruction or a pattern or a stylesheet function it is set to an empty sequence. On completion of an instruction that changes the value, the variable reverts to its previous value.

The value of the current captured substrings is unaffected through calls of `xsl:apply-templates`, `xsl:call-template`, `xsl:apply-imports` or `xsl:next-match`, or by expansion of named attribute sets.

15.3 Examples of Regular Expression Matching

Example: Replacing Characters by Elements

Problem: replace all newline characters in the `abstract` element by empty `br` elements:

Solution:

```
<xsl:analyze-string select="abstract" regex="\n">
  <xsl:matching-substring>
    <br/>
  </xsl:matching-substring>
  <xsl:non-matching-substring>
    <xsl:value-of select="."/>
  </xsl:non-matching-substring>
</xsl:analyze-string>
```

Example: Recognizing non-XML Markup Structure

Problem: replace all occurrences of `[...]` in the `body` by `cite` elements, retaining the content between the square brackets as the content of the new element.

Solution:

```
<xsl:analyze-string select="body" regex="\[(.*?)\]">
  <xsl:matching-substring>
    <cite><xsl:value-of select="regex-group(1)"/></cite>
  </xsl:matching-substring>
  <xsl:non-matching-substring>
    <xsl:value-of select="."/>
  </xsl:non-matching-substring>
</xsl:analyze-string>
```

Note that this simple approach fails if the `body` element contains markup that needs to be retained. In this case it is necessary to apply the regular expression processing to each text node individually. If the `[...]` constructs span multiple text nodes (for example, because there are elements within the square brackets) then it probably becomes necessary to make two or more passes over the data.

Example: Parsing a Date

Problem: the input string contains a date such as `23 March 2002`. Convert it to the form `2002-03-23`.

Solution (with no error handling if the input format is incorrect):

```
<xsl:variable name="months" select="'January', 'February', 'March', ..."/>

<xsl:analyze-string select="normalize-space($input)"
    regex="([0-9]{{1,2}})\s([A-Z][a-z]+)\s([0-9]{{4}})">
    <xsl:matching-substring>
        <xsl:number value="regex-group(3)" format="0001"/>
        <xsl:text>-</xsl:text>
        <xsl:number value="index-of($months, regex-group(2))" format="01"/>
        <xsl:text>-</xsl:text>
        <xsl:number value="regex-group(1)" format="01"/>
    </xsl:matching-substring>
</xsl:analyze-string>
```

Note the use of `normalize-space` to simplify the work done by the regular expression, and the use of doubled curly brackets because the `regex` attribute is an attribute value template.

16 Additional Functions

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

16.1 Multiple Source Documents

```
document($uri-sequence as item()*) as node()*
document($uri-sequence as item()*, $base-node as node()) as node()*
```

The `document` function allows access to XML documents identified by a URI.

The first argument contains a sequence of URI references. The second argument, if present, is a node whose base URI is used to resolve any relative URI references contained in the first argument.

A sequence of absolute URI references is obtained as follows.

- For an item in `$uri-sequence` that is an instance of `xs:string`, `xs:anyURI`, or `xs:untypedAtomic`, the value is cast to `xs:anyURI`. If the resulting URI reference is an absolute URI reference then it is used *as is*. If it is a relative URI reference, then it is resolved against the base URI of `$base-node` if supplied, or against the base URI from the static context otherwise (this will usually be the base URI of the stylesheet module). A relative URI is resolved against a base URI using the rules defined in [RFC3986].
- For an item in `$uri-sequence` that is a node, the node is atomized. The result MUST be a sequence whose items are all instances of `xs:string`, `xs:anyURI`, or `xs:untypedAtomic`. Each of these values is cast to `xs:anyURI`, and if the resulting URI reference is an absolute URI reference then it is used *as is*. If it is a relative URI reference, then it is resolved against the base URI of `$base-node` if supplied, or against the base URI of the node that contained it otherwise.

Note:

The XPath rules for function calling ensure that it is a type error if the supplied value of the second argument is anything other than a single node. If XPath 1.0 compatibility mode is enabled, then a sequence of nodes may be supplied, and the first node in the sequence will be used.

Each of these absolute URI references is then processed as follows. Any fragment identifier that is present in the URI reference is removed, and the resulting absolute URI is cast to a string and then passed to the `doc` ^{FO} function defined in [Functions and Operators]. This returns a document node. If an error occurs during evaluation of the `doc` ^{FO} function, the processor MAY either signal this error in the normal way, or MAY recover by ignoring the failure, in which case the failing URI will not contribute any nodes to the result of the `document` function.

If the URI reference contained no fragment identifier, then this document node is included in the sequence of nodes returned by the `document` function.

If the URI reference contained a fragment identifier, then the fragment identifier is interpreted according to the rules for the media type of the resource representation identified by the URI, and is used to select zero or more nodes that are descendant-or-self nodes of the returned document node. As described in *2.3 Initiating a Transformation*, the media type is available as part of the evaluation context for a transformation.

[ERR XTRE1160] When a URI reference contains a fragment identifier, it is a recoverable dynamic error if the media type is not one that is recognized by the processor, or if the fragment identifier does not conform to the rules for fragment identifiers for that media type, or if the fragment identifier selects something other than a sequence of nodes (for example, if it selects a range of characters within a text node). The optional recovery action is to ignore the fragment identifier and return the document node. The set of media types recognized by a processor is implementation-defined.

Note:

The recovery action here is different from XSLT 1.0

The sequence of nodes returned by the function is in document order, with no duplicates. This order has no necessary relationship to the order in which URIs were supplied in the `$uri-sequence` argument.

Note:

One effect of these rules is that unless XML entities or `xml:base` are used, and provided that the base URI of the stylesheet module is known, `document("")` refers to the document node of the containing stylesheet module (the definitive rules are in [RFC3986]). The XML resource containing the stylesheet module is processed exactly as if it were any other XML document, for example there is no special recognition of `xsl:text` elements, and no special treatment of comments and processing instructions.

16.2 Reading Text Files

```
unparsed-text($href as xs:string?) as xs:string?
unparsed-text($href as xs:string?, $encoding as xs:string) as xs:string?
```

The `unparsed-text` function reads an external resource (for example, a file) and returns its contents as a string.

The `$href` argument MUST be a string in the form of a URI. The URI MUST contain no fragment identifier, and MUST identify a resource that can be read as text. If the URI is a relative URI, then it is resolved relative to the base URI from the static context.

If the value of the `$href` argument is an empty sequence, the function returns an empty sequence.

Note:

If a different base URI is appropriate (for example, when resolving a relative URI read from a source document) then the relative URI should be resolved using the `resolve-uri` ^{FO} function before passing it to the `unparsed-text` function.

The `$encoding` argument, if present, is the name of an encoding. The values for this attribute follow the same rules as for the `encoding` attribute in an XML declaration. The only values which every implementation is REQUIRED to recognize are `utf-8` and `utf-16`.

The encoding of the external resource is determined as follows:

1. external encoding information is used if available, otherwise
2. if the media type of the resource is `text/xml` or `application/xml` (see [RFC2376]), or if it matches the conventions `text/*+xml` or `application/*+xml` (see [RFC3023] and/or its successors), then the encoding is recognized as specified in [XML 1.0], otherwise
3. the value of the `$encoding` argument is used if present, otherwise
4. the processor MAY use implementation-defined heuristics to determine the likely encoding, otherwise
5. UTF-8 is assumed.

Note:

The above rules are chosen for consistency with [XInclude]. Files with an XML media type are treated specially because there are use cases for this function where the retrieved text is to be included as unparsed XML within a CDATA section of a containing document, and because processors are likely to be able to reuse the code that performs encoding detection for XML external entities.

[ERR XTDE1170] It is a non-recoverable dynamic error if a URI contains a fragment identifier, or if it cannot be used to retrieve a resource containing text.

[ERR XTDE1190] It is a non-recoverable dynamic error if a resource contains octets that cannot be decoded into Unicode characters using the specified encoding, or if the resulting characters are not permitted XML characters. This includes the case where the processor does not support the requested encoding.

[ERR XTDE1200] It is a non-recoverable dynamic error if the second argument of the `unparsed-text` function is omitted and the processor cannot infer the encoding using external information and the encoding is not UTF-8.

The result is a string containing the text of the resource retrieved using the URI.

Note:

If the text file contains characters such as `<` and `&`, these will typically be output as `<` and `&` when the string is written to a final result tree and serialized as XML or HTML. If these characters actually represent markup (for example, if the text file contains HTML), then the stylesheet can attempt to write them as markup to the output file using the `disable-output-escaping` attribute of the `xsl:value-of` instruction (see *20.2 Disabling Output Escaping*). Note, however, that implementations are not required to support this feature.

Example: Copying Unparsed HTML Boilerplate

This example attempts to read an HTML file and copy it, as HTML, to the serialized output file:

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

<xsl:template match="/">
  <xsl:value-of select="unparsed-text('header.html', 'iso-8859-1')"
                disable-output-escaping="yes"/>
  <xsl:apply-templates/>
  <xsl:value-of select="unparsed-text('footer.html', 'iso-8859-1')"
                disable-output-escaping="yes"/>
</xsl:template>
```

Example: Splitting an Input File into a Sequence of Lines

Often it is necessary to split a text file into a sequence of lines, representing each line as a string. This can be achieved by using the `unparsed-text` function in conjunction with the XPath `tokenize`[FO] function. For example:

```
<xsl:for-each select="tokenize(unparsed-text($in), '\r?\n')">
 ...
</xsl:for-each>
```

Note that the `unparsed-text` function does not normalize line endings. This example has therefore been written to recognize both Unix and Windows conventions for end-of-line, namely a single newline (#x0A) character or a carriage return / line feed pair (#x0D #x0A).

Because errors in evaluating the `unparsed-text` function are non-recoverable, two functions are provided to allow a stylesheet to determine whether a call with particular arguments would succeed:

```
unparsed-text-available($href as xs:string?) as xs:boolean
```

```
unparsed-text-available($href     as xs:string?,
                        $encoding as xs:string?) as xs:boolean
```

The `unparsed-text-available` function determines whether a call on the `unparsed-text` function with identical arguments would return a string.

If the first argument is an empty sequence, the function returns false. If the second argument is an empty sequence, the function behaves as if the second argument were omitted.

In other cases, the function returns true if a call on `unparsed-text` with the same arguments would succeed, and false if a call on `unparsed-text` with the same arguments would fail with a non-recoverable dynamic error.

Note:

This requires that the `unparsed-text-available` function should actually attempt to read the resource identified by the URI, and check that it is correctly encoded and contains no characters that are invalid in XML. Implementations may avoid the cost of repeating these checks for example by caching the validated contents of the resource, to anticipate a subsequent call on the `unparsed-text` function. Alternatively, implementations may be able to rewrite an expression such as `if (unparsed-text-available(A)) then unparsed-text (A) else ...` to generate a single call internally.

The functions `unparsed-text` and `unparsed-text-available` have the same requirement for stability as the functions `doc` FO and `doc-available` FO defined in [Functions and Operators]. This means that unless the user has explicitly stated a requirement for a reduced level of stability, either of these functions if called twice with the same arguments during the course of a transformation MUST return the same results each time; moreover, the results of a call on `unparsed-text-available` MUST be consistent with the results of a subsequent call on `unparsed-text` with the same arguments.

16.3 Keys

Keys provide a way to work with documents that contain an implicit cross-reference structure. They make it easier to locate the nodes within a document that have a given value for a given attribute or child element, and they provide a hint to the implementation that certain access paths in the document need to be efficient.

16.3.1 The `xsl:key` Declaration

```
<!-- Category: declaration -->
<xsl:key
  name = qname
  match = pattern
  use? = expression
  collation? = uri>
  <!-- Content: sequence-constructor -->
</xsl:key>
```

The `xsl:key` declaration 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 *5.1 Qualified Names*. The `match` attribute is a Pattern; an `xsl:key` element applies to all nodes that match the pattern specified in the `match` attribute.

[DEFINITION: A **key** is defined as a set of `xsl:key` declarations in the stylesheet that share the same name.]

The value of the key may be specified either using the `use` attribute or by means of the contained sequence constructor.

[ERR XTSE1205] It is a static error if an `xsl:key` declaration has a `use` attribute and has non-empty content, or if it has empty content and no `use` attribute.

If the `use` attribute is present, its value is an expression specifying the values of the key. The expression will be evaluated with the node that matches the pattern as the context node. The result of evaluating the expression is atomized.

Similarly, if a sequence constructor is present, it is used to determine the values of the key. The sequence constructor will be evaluated with the node that matches the pattern as the context node. The result of evaluating the sequence constructor is atomized.

[DEFINITION: The expression in the `use` attribute and the sequence constructor within an `xsl:key` declaration are referred to collectively as the **key specifier**. The key specifier determines the values that may be used to find a node using this key.]

Note:

There is no requirement that all the values of a key should have the same type.

The presence of an `xsl:key` declaration makes it easy to find a node that matches the `match` pattern if any of the values of the key specifier (when applied to that node) are known. It also provides a hint to the implementation that access to the nodes by means of these values needs to be efficient (many implementations are likely to construct an index or hash table to achieve this). Note that the key specifier in general returns a sequence of values, and any one of these may be used to locate the node.

Note:

An `xsl:key` declaration is not bound to a specific source document. The source document to which it applies is determined only when the `key` function is used to locate nodes using the key. Keys can be used to locate nodes within any source document (including temporary trees), but each use of the `key` function searches one document only.

The optional `collation` attribute is used only when deciding whether two strings are equal for the purposes of key matching. Specifically, two values `$a` and `$b` are considered equal if the result of the function call `compare($a, $b, $collation)` is zero. The effective collation for an `xsl:key` declaration is the collation specified in its `collation` attribute if present, resolved against the base URI of the `xsl:key` element, or the default collation that is in scope for the `xsl:key` declaration otherwise; the effective collation must be the same for all the `xsl:key` declarations making up a key.

[ERR XTSE1210] It is a static error if the `xsl:key` declaration has a `collation` attribute whose value (after resolving against the base URI) is not a URI recognized by the implementation as referring to a collation.

[ERR XTSE1220] It is a static error if there are several `xsl:key` declarations in the stylesheet with the same key name and different effective collations. Two collations are the same if their URIs are equal under the rules for comparing `xs:anyURI` values, or if the implementation can determine that they are different URIs referring to the same collation.

It is possible to have:

- multiple `xsl:key` declarations with the same name;
- a node that matches the `match` patterns of several different `xsl:key` declarations, whether these have the same key name or different key names;

- a node that returns more than one value from its key specifier;
- a key value that identifies more than one node (the key values for different nodes do not need to be unique).

An `xsl:key` declaration with higher import precedence does not override another of lower import precedence; all the `xsl:key` declarations in the stylesheet are effective regardless of their import precedence.

16.3.2 The key Function

```
key($key-name as xs:string, $key-value as xs:anyAtomicType*) as node()*
```

```
key($key-name  as xs:string,
    $key-value as xs:anyAtomicType*,
    $top       as node()) as node()*
```

The `key` function does for keys what the `id` FO function does for IDs.

The `$key-name` argument specifies the name of the key. The value of the argument MUST be a lexical QName, which is expanded as described in *5.1 Qualified Names*.

[ERR XTDE1260] It is a non-recoverable dynamic error if the value is not a valid QName, or if there is no namespace declaration in scope for the prefix of the QName, or if the name obtained by expanding the QName is not the same as the expanded name of any `xsl:key` declaration in the stylesheet. If the processor is able to detect the error statically (for example, when the argument is supplied as a string literal), then the processor MAY optionally signal this as a static error.

The `$key-value` argument to the `key` function is considered as a sequence. The set of requested key values is formed by atomizing the supplied value of the argument, using the standard function conversion rules. Each of the resulting atomic values is considered as a requested key value. The result of the function is a sequence of nodes, in document order and with duplicates removed, comprising those nodes in the selected subtree (see below) that are matched by an `xsl:key` declaration whose name is the same as the supplied key name, where the result of evaluating the key specifier contains a value that is equal to one of these requested key values, under the rules appropriate to the XPath `eq` operator for the two values in question, using the `collation` attributes of the `xsl:key` declaration when comparing strings. No error is reported if two values are encountered that are not comparable; they are regarded for the purposes of this function as being not equal.

> **Note:**
>
> Under the rules for the `eq` operator, untyped atomic values are converted to strings, not to the type of the other operand. This means, for example, that if the expression in the `use` attribute returns a date, supplying an untyped atomic value in the call to the `key` function will return an empty sequence.

If the second argument is an empty sequence, the result of the function will be an empty sequence.

Different rules apply when backwards compatible behavior is enabled. Specifically, if any of the `xsl:key` elements in the definition of the key enables backwards compatible behavior, then the value of the key specifier and the value of the second argument of the `key` function are both converted after atomization to a sequence of strings, by applying a cast to each item in the sequence, before performing the comparison.

The third argument is used to identify the selected subtree. If the argument is present, the selected subtree is the set of nodes that have *$top* as an ancestor-or-self node. If the argument is omitted, the selected subtree is the document containing the context node. This means that the third argument effectively defaults to `/`.

[ERR XTDE1270] It is a non-recoverable dynamic error to call the `key` function with two arguments if there is no context node, or if the root of the tree containing the context node is not a document node; or to call the function with three arguments if the root of the tree containing the node supplied in the third argument is not a document node.

The result of the `key` function can be described more specifically as follows. The result is a sequence containing every node *$N* that satisfies the following conditions:

- `$N/ancestor-or-self::node() intersect $top` is non-empty. (If the third argument is omitted, `$top` defaults to `/`)
- *$N* matches the pattern specified in the `match` attribute of an `xsl:key` declaration whose `name` attribute matches the name specified in the `$key-name` argument.
- When the key specifier of that `xsl:key` declaration is evaluated with a singleton focus based on *$N*, the atomized value of the resulting sequence includes a value that compares equal to at least one item in the atomized value of the sequence supplied as `$key-value`, under the rules of the `eq` operator with the collation selected as described above.

The sequence returned by the `key` function will be in document order, with duplicates (that is, nodes having the same identity) removed.

Example: Using a Key to Follow Cross-References

For example, given a declaration

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

an expression `key("idkey",@ref)` will return the same nodes 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 context node contains no whitespace.

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

```
<prototype name="sqrt" return-type="xs:double">
  <arg type="xs:double"/>
</prototype>
```

and a `function` element to refer to function names

```
<function>sqrt</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>
```

When called with two arguments, the `key` function always returns nodes that are in the same document as the context node. To retrieve a node from any other document, it is necessary either to change the context node, or to supply a third argument.

Example: Using Keys to Reference other Documents

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

Note:

This relies on the ability in XPath 2.0 to have a function call on the right-hand side of the `/` operator in a path expression.

The following code would also work:

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

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

16.4 Number Formatting

```
format-number($value as numeric?, $picture as xs:string) as xs:string
```

```
format-number($value               as numeric?,
              $picture             as xs:string,
              $decimal-format-name as xs:string) as xs:string
```

The `format-number` function formats `$value` as a string using the picture string specified by the `$picture` argument and the decimal-format named by the `$decimal-format-name` argument, or the default decimal-format, if there is no `$decimal-format-name` argument. The syntax of the picture string is described in *16.4.2 Processing the Picture String*.

The `$value` argument may be of any numeric data type (`xs:double`, `xs:float`, `xs:decimal`, or their subtypes including `xs:integer`). Note that if an `xs:decimal` is supplied, it is not automatically promoted to an `xs:double`, as such promotion can involve a loss of precision.

If the supplied value of the `$value` argument is an empty sequence, the function behaves as if the supplied value were the `xs:double` value `NaN`.

The value of `$decimal-format-name` MUST be a lexical QName, which is expanded as described in *5.1 Qualified Names*. The result of the function is the formatted string representation of the supplied number.

[ERR XTDE1280] It is a non-recoverable dynamic error if the name specified as the `$decimal-format-name` argument is not a valid QName, or if its prefix has not been declared in an in-scope namespace declaration, or if the stylesheet does not contain a declaration of a decimal-format

with a matching expanded-QName. If the processor is able to detect the error statically (for example, when the argument is supplied as a string literal), then the processor MAY optionally signal this as a static error.

16.4.1 Defining a Decimal Format

```
<!-- Category: declaration -->
<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 controls the interpretation of a picture string used by the `format-number` function.

A stylesheet may contain multiple `xsl:decimal-format` declarations and may include or import stylesheet modules that also contain `xsl:decimal-format` declarations. The name of an `xsl:decimal-format` declaration is the value of its `name` attribute, if any.

[DEFINITION: All the `xsl:decimal-format` declarations in a stylesheet that share the same name are grouped into a named **decimal format**; those that have no name are grouped into a single unnamed decimal format.]

If a stylesheet does not contain a declaration of the unnamed decimal format, a declaration equivalent to an `xsl:decimal-format` element with no attributes is implied.

The attributes of the `xsl:decimal-format` declaration establish values for a number of variables used as input to the algorithm followed by the `format-number` function. An outline of the purpose of each attribute is given below; however, the definitive explanations are given later, as part of the description of this algorithm.

For any named decimal format, the effective value of each attribute is taken from an `xsl:decimal-format` declaration that has that name, and that specifies an explicit value for the required attribute. If there is no such declaration, the default value of the attribute is used. If there is more than one such declaration, the one with highest import precedence is used.

For any unnamed decimal format, the effective value of each attribute is taken from an `xsl:decimal-format` declaration that is unnamed, and that specifies an explicit value for the required attribute. If there is no such declaration, the default value of the attribute is used. If there is more than one such declaration, the one with highest import precedence is used.

[ERR XTSE1290] It is a static error if a named or unnamed decimal format contains two conflicting values for the same attribute in different `xsl:decimal-format` declarations having the same import precedence, unless there is another definition of the same attribute with higher import precedence.

The following attributes control the interpretation of characters in the picture string supplied to the `format-number` function, and also specify characters that may appear in the result of formatting the number. In each case the value MUST be a single character [see ERR XTSE0020].

- `decimal-separator` specifies the character used for the *decimal-separator-sign*; the default value is the period character (`.`)
- `grouping-separator` specifies the character used for the *grouping-sign*, which is typically used as a thousands separator; the default value is the comma character (`,`)
- `percent` specifies the character used for the *percent-sign*; the default value is the percent character (`%`)
- `per-mille` specifies the character used for the *per-mille-sign*; the default value is the Unicode per-mille character (#x2030)
- `zero-digit` specifies the character used for the *digit-zero-sign*; the default value is the digit zero (`0`). This character MUST be a digit (category Nd in the Unicode property database), and it MUST have the numeric value zero. This attribute implicitly defines the Unicode character that is used to represent each of the values 0 to 9 in the final result string: Unicode is organized so that each set of decimal digits forms a contiguous block of characters in numerical sequence.

[ERR XTSE1295] It is a static error if the character specified in the `zero-digit` attribute is not a digit or is a digit that does not have the numeric value zero.

The following attributes control the interpretation of characters in the picture string supplied to the `format-number` function. In each case the value MUST be a single character [see ERR XTSE0020].

- `digit` specifies the character used for the *digit-sign* in the picture string; the default value is the number sign character (`#`)
- `pattern-separator` specifies the character used for the *pattern-separator-sign*, which separates positive and negative sub-pictures in a picture string; 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 for the *infinity-symbol*; the default value is the string `Infinity`
- `NaN` specifies the string used for the *NaN-symbol*, which is used to represent the value NaN (not-a-number); the default value is the string `NaN`
- `minus-sign` specifies the character used for the *minus-symbol*; the default value is the hyphen-minus character (`-`, #x2D). The value MUST be a single character.

[ERR XTSE1300] It is a static error if, for any named or unnamed decimal format, the variables representing characters used in a picture string do not each have distinct values. These variables are *decimal-separator-sign*, *grouping-sign*, *percent-sign*, *per-mille-sign*, *digit-zero-sign*, *digit-sign*, and *pattern-separator-sign*.

16.4.2 Processing the Picture String

[DEFINITION: The formatting of a number is controlled by a **picture string**. The picture string is a sequence of characters, in which the characters assigned to the variables *decimal-separator-sign*, *grouping-sign*, *zero-digit-sign*, *digit-sign* and *pattern-separator-sign* are

classified as active characters, and all other characters (including the *percent-sign* and *per-mille-sign*) are classified as passive characters.]

The *integer part* of the sub-picture is defined as the part that appears to the left of the *decimal-separator-sign* if there is one, or the entire sub-picture otherwise. The *fractional part* of the sub-picture is defined as the part that appears to the right of the *decimal-separator-sign* if there is one; it is a zero-length string otherwise.

[ERR XTDE1310] The picture string MUST conform to the following rules. It is a non-recoverable dynamic error if the picture string does not satisfy these rules.

Note that in these rules the words "preceded" and "followed" refer to characters anywhere in the string, they are not to be read as "immediately preceded" and "immediately followed".

- A picture-string consists either of a sub-picture, or of two sub-pictures separated by a *pattern-separator-sign*. A picture-string MUST NOT contain more than one *pattern-separator-sign*. If the picture-string contains two sub-pictures, the first is used for positive values and the second for negative values.
- A sub-picture MUST NOT contain more than one *decimal-separator-sign*.
- A sub-picture MUST NOT contain more than one *percent-sign* or *per-mille-sign*, and it MUST NOT contain one of each.
- A sub-picture MUST contain at least one *digit-sign* or *zero-digit-sign*.
- A sub-picture MUST NOT contain a passive character that is preceded by an active character and that is followed by another active character.
- A sub-picture MUST NOT contain a *grouping-separator-sign* adjacent to a *decimal-separator-sign*.
- The integer part of a sub-picture MUST NOT contain a *zero-digit-sign* that is followed by a *digit-sign*. The fractional part of a sub-picture MUST NOT contain a *digit-sign* that is followed by a *zero-digit-sign*.

The evaluation of the `format-number` function is described below in two phases, an analysis phase and a formatting phase. The analysis phase takes as its inputs the picture string and the variables derived from the relevant `xsl:decimal-format` declaration, and produces as its output a number of variables with defined values. The formatting phase takes as its inputs the number to be formatted and the variables produced by the analysis phase, and produces as its output a string containing a formatted representation of the number.

Note:

Numbers will always be formatted with the most significant digit on the left.

16.4.3 Analysing the Picture String

This phase of the algorithm analyses the picture string and the attribute settings of the `xsl:decimal-format` declaration, and has the effect of setting the values of various variables, which are used in the subsequent formatting phase. These variables are listed below. Each is shown with its initial setting and its data type.

Several variables are associated with each sub-picture. If there are two sub-pictures, then these rules are applied to one sub-picture to obtain the values that apply to positive numbers, and to the other to obtain the values that apply to negative numbers. If there is only one sub-picture, then the values for both cases are derived from this sub-picture.

The variables are as follows:

- The *integer-part-grouping-positions* is a sequence of integers representing the positions of grouping separators within the integer part of the sub-picture. For each *grouping-separator-sign* that appears within the integer part of the sub-picture, this sequence contains an integer that is equal to the total number of *digit-sign* and *zero-digit-sign* characters that appear within the integer part of the sub-picture and to the right of the *grouping-separator-sign*. In addition, if these *integer-part-grouping-positions* are at regular intervals (that is, if they form a sequence *N*, *2N*, *3N*, ... for some integer value *N*, including the case where there is only one number in the list), then the sequence contains all integer multiples of *N* as far as necessary to accommodate the largest possible number.
- The *minimum-integer-part-size* is an integer indicating the minimum number of digits that will appear to the left of the *decimal-separator-sign*. It is normally set to the number of *zero-digit-sign* characters found in the integer part of the sub-picture. But if the sub-picture contains no *zero-digit-sign* and no *decimal-separator-sign*, it is set to one.

  Note:

  There is no maximum integer part size. All significant digits in the integer part of the number will be displayed, even if this exceeds the number of *digit-sign* and *zero-digit-sign* characters in the subpicture.
- The *prefix* is set to contain all passive characters in the sub-picture to the left of the leftmost active character. If the picture string contains only one sub-picture, the *prefix* for the negative sub-picture is set by concatenating the *minus-sign* character and the *prefix* for the positive sub-picture (if any), in that order.
- The *fractional-part-grouping-positions* is a sequence of integers representing the positions of grouping separators within the fractional part of the sub-picture. For each *grouping-separator-sign* that appears within the fractional part of the sub-picture, this sequence contains an integer that is equal to the total number of *digit-sign* and *zero-digit-sign* characters that appear within the fractional part of the sub-picture and to the left of the *grouping-separator-sign*.
- The *minimum-fractional-part-size* is set to the number of *zero-digit-sign* characters found in the fractional part of the sub-picture.
- The *maximum-fractional-part-size* is set to the total number of *digit-sign* and *zero-digit-sign* characters found in the fractional part of the sub-picture.
- The *suffix* is set to contain all passive characters to the right of the rightmost active character in the fractional part of the sub-picture.

Note:

If there is only one sub-picture, then all variables for positive numbers and negative numbers will be the same, except for *prefix*: the prefix for negative numbers will be preceded by the *minus-sign* character.

16.4.4 Formatting the Number

This section describes the second phase of processing of the `format-number` function. This phase takes as input a number to be formatted (referred to as the *input number*), and the variables set up by analysing the `xsl:decimal-format` declaration and the picture string, as described above. The result of this phase is a string, which forms the return value of the `format-number` function.

The algorithm for this second stage of processing is as follows:

1. If the input number is NaN (not a number), the result is the specified *NaN-symbol* (with no *prefix* or *suffix*).
2. In the rules below, the positive sub-picture and its associated variables are used if the input number is positive, and the negative sub-picture and its associated variables are used otherwise. Negative zero is taken as negative, positive zero as positive.
3. If the input number is positive or negative infinity, the result is the concatenation of the appropriate *prefix*, the *infinity-symbol*, and the appropriate *suffix*.
4. If the sub-picture contains a *percent-sign*, the number is multiplied by 100. If the sub-picture contains a *per-mille-sign*, the number is multiplied by 1000. The resulting number is referred to below as the *adjusted number*.
5. The *adjusted number* is converted (if necessary) to an `xs:decimal` value, using an implementation of `xs:decimal` that imposes no limits on the `totalDigits` or `fractionDigits` facets. If there are several such values that are numerically equal to the *adjusted number* (bearing in mind that if the *adjusted number* is an `xs:double` or `xs:float`, the comparison will be done by converting the decimal value back to an `xs:double` or `xs:float`), the one that is chosen SHOULD be one with the smallest possible number of digits not counting leading or trailing zeroes (whether significant or insignificant). For example, 1.0 is preferred to 0.9999999999, and 100000000 is preferred to 100000001. This value is then rounded so that it uses no more than `maximum-fractional-part-size` digits in its fractional part. The *rounded number* is defined to be the result of converting the *adjusted number* to an `xs:decimal` value, as described above, and then calling the function `round-half-to-even`[FO] with this converted number as the first argument and the `maximum-fractional-part-size` as the second argument, again with no limits on the `totalDigits` or `fractionDigits` in the result.
6. The absolute value of the *rounded number* is converted to a string in decimal notation, with no insignificant leading or trailing zeroes, using the characters implied by the choice of *zero-digit-sign* to represent the ten decimal digits, and the *decimal-separator-sign* to separate the integer part and the fractional part. (The value zero will at this stage be represented by a *decimal-separator-sign* on its own.)
7. If the number of digits to the left of the *decimal-separator-sign* is less than *minimum-integer-part-size*, leading *zero-digit-sign* characters are added to pad out to that size.
8. If the number of digits to the right of the *decimal-separator-sign* is less than *minimum-fractional-part-size*, trailing *zero-digit-sign* characters are added to pad out to that size.
9. For each integer *N* in the *integer-part-grouping-positions* list, a *grouping-separator-sign* character is inserted into the string immediately after that digit that appears in the integer part of the number and has *N* digits between it and the *decimal-separator-sign*, if there is such a digit.
10. For each integer *N* in the *fractional-part-grouping-positions* list, a *grouping-separator-sign* character is inserted into the string immediately before that digit that appears in the fractional part of the number and has *N* digits between it and the *decimal-separator-sign*, if there is such a digit.
11. If there is no *decimal-separator-sign* in the sub-picture, or if there are no digits to the right of the *decimal-separator-sign* character in the string, then the *decimal-separator-sign* character is removed from the string (it will be the rightmost character in the string).
12. The result of the function is the concatenation of the appropriate *prefix*, the string conversion of the number as obtained above, and the appropriate *suffix*.

16.5 Formatting Dates and Times

Three functions are provided to represent dates and times as a string, using the conventions of a selected calendar, language, and country. Each has two variants.

```
format-dateTime($value    as xs:dateTime?,
                $picture  as xs:string,
                $language as xs:string?,
                $calendar as xs:string?,
                $country  as xs:string?) as xs:string?
```

```
format-dateTime($value  as xs:dateTime?, $picture as xs:string) as xs:string?
```

```
format-date($value    as xs:date?,
            $picture  as xs:string,
            $language as xs:string?,
            $calendar as xs:string?,
            $country  as xs:string?) as xs:string?
```

```
format-date($value as xs:date?, $picture as xs:string) as xs:string?
```

```
format-time($value    as xs:time?,
            $picture  as xs:string,
            $language as xs:string?,
            $calendar as xs:string?,
            $country  as xs:string?) as xs:string?
```

```
format-time($value as xs:time?, $picture as xs:string) as xs:string?
```

The `format-dateTime`, `format-date`, and `format-time` functions format `$value` as a string using the picture string specified by the `$picture` argument, the calendar specified by the `$calendar` argument, the language specified by the `$language` argument, and the country specified by the `$country` argument. The result of the function is the formatted string representation of the supplied `dateTime`, `date`, or `time` value.

[DEFINITION: The three functions `format-date`, `format-time`, and `format-dateTime` are referred to collectively as the **date formatting functions**.]

If `$value` is the empty sequence, the empty sequence is returned.

Calling the two-argument form of each of the three functions is equivalent to calling the five-argument form with each of the last three arguments set to an empty sequence.

For details of the `language`, `calendar`, and `country` arguments, see *16.5.2 The Language, Calendar, and Country Arguments*.

In general, the use of an invalid `picture`, `language`, `calendar`, or `country` argument is classified as a non-recoverable dynamic error. By contrast, use of an option in any of these arguments that is valid but not supported by the implementation is not an error, and in these cases the implementation is required to output the value in a fallback representation.

16.5.1 The Picture String

The picture consists of a sequence of variable markers and literal substrings. A substring enclosed in square brackets is interpreted as a

variable marker; substrings not enclosed in square brackets are taken as literal substrings. The literal substrings are optional and if present are rendered unchanged, including any whitespace. If an opening or closing square bracket is required within a literal substring, it MUST be doubled. The variable markers are replaced in the result by strings representing aspects of the date and/or time to be formatted. These are described in detail below.

A variable marker consists of a component specifier followed optionally by one or two presentation modifiers and/or optionally by a width modifier. Whitespace within a variable marker is ignored.

The *component specifier* indicates the component of the date or time that is required, and takes the following values:

Specifier	Meaning	Default Presentation Modifier
Y	year (absolute value)	1
M	month in year	1
D	day in month	1
d	day in year	1
F	day of week	n
W	week in year	1
w	week in month	1
H	hour in day (24 hours)	1
h	hour in half-day (12 hours)	1
P	am/pm marker	n
m	minute in hour	01
s	second in minute	01
f	fractional seconds	1
Z	timezone as a time offset from UTC, or if an alphabetic modifier is present the conventional name of a timezone (such as PST)	1
z	timezone as a time offset using GMT, for example GMT+1	1
C	calendar: the name or abbreviation of a calendar name	n
E	era: the name of a baseline for the numbering of years, for example the reign of a monarch	n

[ERR XTDE1340] It is a non-recoverable dynamic error if the syntax of the picture is incorrect.

[ERR XTDE1350] It is a non-recoverable dynamic error if a component specifier within the picture refers to components that are not available in the given type of `$value`, for example if the picture supplied to the `format-time` refers to the year, month, or day component.

It is not an error to include a timezone component when the supplied value has no timezone. In these circumstances the timezone component will be ignored.

The first *presentation modifier* indicates the style in which the value of a component is to be represented. Its value may be either:

- any format token permitted in the `format` string of the `xsl:number` instruction (see *12 Numbering*), indicating that the value of the component is to be output numerically using the specified number format (for example, `1`, `01`, `i`, `I`, `w`, `W`, or `Ww`) or
- the format token `n`, `N`, or `Nn`, indicating that the value of the component is to be output by name, in lower-case, upper-case, or title-case respectively. Components that can be output by name include (but are not limited to) months, days of the week, timezones, and eras. If the processor cannot output these components by name for the chosen calendar and language then it must use an implementation-defined fallback representation.

If the implementation does not support the use of the requested format token, it MUST use the default presentation modifier for that component.

If the first presentation modifier is present, then it may optionally be followed by a second presentation modifier as follows:

Modifier	Meaning
t	traditional numbering. This has the same meaning as `letter-value="traditional"` in `xsl:number`.
o	ordinal form of a number, for example `8th` or `8°`. The actual representation of the ordinal form of a number may depend not only on the language, but also on the grammatical context (for example, in some languages it must agree in gender).

Note:

Although the formatting rules are expressed in terms of the rules for format tokens in `xsl:number`, the formats actually used may be specialized to the numbering of date components where appropriate. For example, in Italian, it is conventional to use an ordinal number (`primo`) for the first day of the month, and cardinal numbers (`due, tre, quattro ...`) for the remaining days. A processor may therefore use this convention to number days of the month, ignoring the presence or absence of the ordinal presentation modifier.

Whether or not a presentation modifier is included, a width modifier may be supplied. This indicates the number of characters or digits to be included in the representation of the value.

The width modifier, if present, is introduced by a comma. It takes the form:

```
,  min-width ("-" max-width)?
```

where `min-width` is either an unsigned integer indicating the minimum number of characters to be output, or `*` indicating that there is no explicit minimum, and `max-width` is either an unsigned integer indicating the maximum number of characters to be output, or `*` indicating that there is no explicit maximum; if `max-width` is omitted then `*` is assumed. Both integers, if present, MUST be greater than zero.

A format token containing leading zeroes, such as `001`, sets the minimum and maximum width to the number of digits appearing in the format token; if a width modifier is also present, then the width modifier takes precedence.

Note:

A format token consisting of a one-digit on its own, such as `1`, does not constrain the number of digits in the output. In the case of fractional seconds in particular, `[f001]` requests three decimal digits, `[f01]` requests two digits, but `[f1]` will produce an implementation-defined number of digits. If exactly one digit is required, this can be achieved using the component specifier `[f1,1-1]`.

If the minumum and maximum width are unspecified, then the output uses as many characters as are required to represent the value of the component without truncation and without padding: this is referred to below as the *full representation* of the value.

If the full representation of the value exceeds the specified maximum width, then the processor SHOULD attempt to use an alternative shorter representation that fits within the maximum width. Where the presentation modifier is `N`, `n`, or `Nn`, this is done by abbreviating the name, using either conventional abbreviations if available, or crude right-truncation if not. For example, setting `max-width` to `4` indicates that four-letter abbreviations SHOULD be used, though it would be acceptable to use a three-letter abbreviation if this is in conventional use. (For example, "Tuesday" might be abbreviated to "Tues", and "Friday" to "Fri".) In the case of the year component, setting `max-width` requests omission of high-order digits from the year, for example, if `max-width` is set to `2` then the year 2003 will be output as `03`. In the case of the fractional seconds component, the value is rounded to the specified size as if by applying the function `round-half-to-even(fractional-seconds, max-width)`. If no mechanism is available for fitting the value within the specified maximum width (for example, when roman numerals are used), then the value SHOULD be output in its full representation.

If the full representation of the value is shorter than the specified minimum width, then the processor SHOULD pad the value to the specified width. For decimal representations of numbers, this SHOULD be done by prepending zero digits from the appropriate set of digit characters, or appending zero digits in the case of the fractional seconds component. In other cases, it SHOULD be done by appending spaces.

16.5.2 The Language, Calendar, and Country Arguments

The set of languages, calendars, and countries that are supported in the date formatting functions is implementation-defined. When any of these arguments is omitted or is an empty sequence, an implementation-defined default value is used.

If the fallback representation uses a different calendar from that requested, the output string MUST be prefixed with `[Calendar: X]` where `X` identifies the calendar actually used. The string `Calendar` SHOULD be localized using the requested language if available. If the fallback representation uses a different language from that requested, the output string should be prefixed with `[Language: Y]` where `Y` identifies the language actually used. The string `Language` MAY be localized in an implementation-dependent way. If a particular component of the value cannot be output in the requested format, it SHOULD be output in the default format for that component.

The `language` argument specifies the language to be used for the result string of the function. The value of the argument MUST be either the empty sequence or a value that would be valid for the `xml:lang` attribute (see [XML]). Note that this permits the identification of sublanguages based on country codes (from [ISO 3166-1]) as well as identification of dialects and of regions within a country.

If the `language` argument is omitted or is set to an empty sequence, or if it is set to an invalid value or a value that the implementation does not recognize, then the processor uses an implementation-defined language.

The language is used to select the appropriate language-dependent forms of:

names (for example, of months)
numbers expressed as words or as ordinals (`twenty, 20th, twentieth`)
hour convention (0-23 vs 1-24, 0-11 vs 1-12)
first day of week, first week of year

Where appropriate this choice may also take into account the value of the `country` argument, though this SHOULD not be used to override the language or any sublanguage that is specified as part of the `language` argument.

The choice of the names and abbreviations used in any given language is implementation-defined. For example, one implementation might abbreviate July as `Jul` while another uses `Jly`. In German, one implementation might represent Saturday as `Samstag` while another uses `Sonnabend`. Implementations MAY provide mechanisms allowing users to control such choices.

Where ordinal numbers are used, the selection of the correct representation of the ordinal (for example, the linguistic gender) MAY depend on the component being formatted and on its textual context in the picture string.

The `calendar` attribute specifies that the `dateTime`, `date`, or `time` supplied in the `$value` argument MUST be converted to a value in the specified calendar and then converted to a string using the conventions of that calendar.

A calendar value MUST be a valid QName. If the QName does not have a prefix, then it identifies a calendar with the designator specified below. If the QName has a prefix, then the QName is expanded into an expanded-QName as described in *5.1 Qualified Names*; the expanded-QName identifies the calendar; the behavior in this case is implementation-defined.

If the calendar attribute is omitted an implementation-defined value is used.

Note:

The calendars listed below were known to be in use during the last hundred years. Many other calendars have been used in the past.

This specification does not define any of these calendars, nor the way that they map to the value space of the `xs:date` data type in [XML Schema Part 2]. There may be ambiguities when dates are recorded using different calendars. For example, the start of a new day is not simultaneous in different calendars, and may also vary geographically (for example, based on the time of sunrise or sunset). Translation of dates is therefore more reliable when the time of day is also known, and when the geographic location is known. When translating dates between one calendar and another, the processor may take account of the values of the `country` and/or `language`

arguments, with the `country` argument taking precedence.

Information about some of these calendars, and algorithms for converting between them, may be found in [Calendrical Calculations].

Designator	Calendar
AD	Anno Domini (Christian Era)
AH	Anno Hegirae (Muhammedan Era)
AME	Mauludi Era (solar years since Mohammed's birth)
AM	Anno Mundi (Jewish Calendar)
AP	Anno Persici
AS	Aji Saka Era (Java)
BE	Buddhist Era
CB	Cooch Behar Era
CE	Common Era
CL	Chinese Lunar Era
CS	Chula Sakarat Era
EE	Ethiopian Era
FE	Fasli Era
ISO	ISO 8601 calendar
JE	Japanese Calendar
KE	Khalsa Era (Sikh calendar)
KY	Kali Yuga
ME	Malabar Era
MS	Monarchic Solar Era
NS	Nepal Samwat Era
OS	Old Style (Julian Calendar)
RS	Rattanakosin (Bangkok) Era
SE	Saka Era
SH	Mohammedan Solar Era (Iran)
SS	Saka Samvat
TE	Tripurabda Era
VE	Vikrama Era
VS	Vikrama Samvat Era

At least one of the above calendars MUST be supported. It is implementation-defined which calendars are supported.

The ISO 8601 calendar ([ISO 8601]), which is included in the above list and designated `ISO`, is very similar to the Gregorian calendar designated `AD`, but it differs in several ways. The ISO calendar is intended to ensure that date and time formats can be read easily by other software, as well as being legible for human users. The ISO calendar prescribes the use of particular numbering conventions as defined in ISO 8601, rather than allowing these to be localized on a per-language basis. In particular it provides a numeric 'week date' format which identifies dates by year, week of the year, and day in the week; in the ISO calendar the days of the week are numbered from 1 (Monday) to 7 (Sunday), and week 1 in any calendar year is the week (from Monday to Sunday) that includes the first Thursday of that year. The numeric values of the components year, month, day, hour, minute, and second are the same in the ISO calendar as the values used in the lexical representation of the date and time as defined in [XML Schema Part 2]. The era ("E" component) with this calendar is either a minus sign (for negative years) or a zero-length string (for positive years). For dates before 1 January, AD 1, year numbers in the ISO and AD calendars are off by one from each other: ISO year 0000 is 1 BC, -0001 is 2 BC, etc.

Note:

The value space of the date and time data types, as defined in XML Schema, is based on absolute points in time. The lexical space of these data types defines a representation of these absolute points in time using the proleptic Gregorian calendar, that is, the modern Western calendar extrapolated into the past and the future; but the value space is calendar-neutral. The date formatting functions produce a representation of this absolute point in time, but denoted in a possibly different calendar. So, for example, the date whose lexical representation in XML Schema is `1502-01-11` (the day on which Pope Gregory XIII was born) might be formatted using the Old Style (Julian) calendar as `1 January 1502`. This reflects the fact that there was at that time a ten-day difference between the two calendars. It would be incorrect, and would produce incorrect results, to represent this date in an element or attribute of type `xs:date` as `1502-01-01`, even though this might reflect the way the date was recorded in contemporary documents.

When referring to years occurring in antiquity, modern historians generally use a numbering system in which there is no year zero (the year before 1 CE is thus 1 BCE). This is the convention that SHOULD be used when the requested calendar is OS (Julian) or AD (Gregorian). When the requested calendar is ISO, however, the conventions of ISO 8601 SHOULD be followed: here the year before +0001 is numbered zero. In [XML Schema Part 2] (version 1.0), the value space for `xs:date` and `xs:dateTime` does not include a year zero: however, a future edition is expected to endorse the ISO 8601 convention. This means that the date on which Julius Caesar was

assassinated has the ISO 8601 lexical representation -0043-03-13, but will be formatted as 15 March 44 BCE in the Julian calendar or 13 March 44 BCE in the Gregorian calendar (dependant on the chosen localization of the names of months and eras).

The intended use of the `country` argument is to identify the place where an event represented by the `dateTime`, `date`, or `time` supplied in the `$value` argument took place or will take place. If the value is supplied, and is not the empty sequence, then it SHOULD be a country code defined in [ISO 3166-1]. Implementations MAY also allow the use of codes representing subdivisions of a country from ISO 3166-2, or codes representing formerly used names of countries from ISO 3166-3. This argument is not intended to identify the location of the user for whom the date or time is being formatted; that should be done by means of the `language` attribute. This information MAY be used to provide additional information when converting dates between calendars or when deciding how individual components of the date and time are to be formatted. For example, different countries using the Old Style (Julian) calendar started the new year on different days, and some countries used variants of the calendar that were out of synchronization as a result of differences in calculating leap years. The geographical area identified by a country code is defined by the boundaries as they existed at the time of the date to be formatted, or the present-day boundaries for dates in the future.

16.5.3 Examples of Date and Time Formatting

Example: Gregorian Calendar

The following examples show a selection of dates and times and the way they might be formatted. These examples assume the use of the Gregorian calendar as the default calendar.

Required Output	Expression
2002-12-31	format-date($d, "[Y0001]-[M01]-[D01]")
12-31-2002	format-date($d, "[M]-[D]-[Y]")
31-12-2002	format-date($d, "[D]-[M]-[Y]")
31 XII 2002	format-date($d, "[D1] [MI] [Y]")
31st December, 2002	format-date($d, "[D1o] [MNn], [Y]", "en", (), ())
31 DEC 2002	format-date($d, "[D01] [MN,*-3] [Y0001]", "en", (), ())
December 31, 2002	format-date($d, "[MNn] [D], [Y]", "en", (), ())
31 Dezember, 2002	format-date($d, "[D] [MNn], [Y]", "de", (), ())
Tisdag 31 December 2002	format-date($d, "[FNn] [D] [MNn] [Y]", "sv", (), ())
[2002-12-31]	format-date($d, "[[[Y0001]-[M01]-[D01]]]")
Two Thousand and Three	format-date($d, "[YWw]", "en", (), ())
einunddreißigste Dezember	format-date($d, "[Dwo] [MNn]", "de", (), ())
3:58 PM	format-time($t, "[h]:[m01] [PN]", "en", (), ())
3:58:45 pm	format-time($t, "[h]:[m01]:[s01] [Pn]", "en", (), ())
3:58:45 PM PDT	format-time($t, "[h]:[m01]:[s01] [PN] [ZN,*-3]", "en", (), ())
3:58:45 o'clock PM PDT	format-time($t, "[h]:[m01]:[s01] o'clock [PN] [ZN,*-3]", "en")
15:58	format-time($t,"[H01]:[m01]")
15:58:45.762	format-time($t,"[H01]:[m01]:[s01].[f001]")
15:58:45 GMT+02:00	format-time($t,"[H01]:[m01]:[s01] [z]", "en", (), ())
15.58 Uhr GMT+02:00	format-time($t,"[H01]:[m01] Uhr [z]", "de", (), ())
3.58pm on Tuesday, 31st December	format-dateTime($dt, "[h].[m01][Pn] on [FNn], [D1o] [MNn]")
12/31/2002 at 15:58:45	format-dateTime($dt, "[M01]/[D01]/[Y0001] at [H01]:[m01]:[s01]")

Example: Non-Gregorian Calendars

The following examples use calendars other than the Gregorian calendar.

These examples use non-Latin characters which might not display correctly in all browsers, depending on the system configuration.

Description	Request	Result
Islamic	`format-date($d, "[D&#x0661;] [Mn] [Y&#x0661;]", "Islamic", "ar", "AH", ())`	٢٦ شوّال ١٤٢٣
Jewish (with Western numbering)	`format-date($d, "[D] [Mn] [Y]", "he", "AM", ())`	26 טבת 5763
Jewish (with traditional numbering)	`format-date($d, "[D&#x05D0;t] [Mn] [Y&#x05D0;t]", "he", "AM", ())`	כ״ו טבת תשס״ג
Julian (Old Style)	`format-date($d, "[D] [MNn] [Y]", "en", "OS", ())`	18 December 2002
Thai	`format-date($d, "[D&#x0E51;] [Mn] [Y&#x0E51;]", "th", "BE", ())`	๓๑ ธันวาคม ๒๕๔๕

16.6 Miscellaneous Additional Functions

16.6.1 current

```
current() as item()
```

The `current` function, used within an XPath expression, returns the item that was the context item at the point where the expression was invoked from the XSLT stylesheet. This is referred to as the current item. For an outermost expression (an expression not occurring within another expression), the current item is always the same as the context item. Thus,

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

means the same as

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

However, within square brackets, or on the right-hand side of the `/` operator, the current item is generally different from the context item.

Example: Using the `current` Function

For example,

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

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

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

which means the same as

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

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

If the `current` function is used within a pattern, its value is the node that is being matched against the pattern.

[ERR XTDE1360] If the `current` function is evaluated within an expression that is evaluated when the context item is undefined, a non-recoverable dynamic error occurs.

16.6.2 unparsed-entity-uri

```
unparsed-entity-uri($entity-name as xs:string) as xs:anyURI
```

The `unparsed-entity-uri` function returns the URI of the unparsed entity whose name is given by the value of the `$entity-name` argument, in the document containing the context node. It returns the zero-length `xs:anyURI` if there is no such entity. This function maps to the `dm:unparsed-entity-system-id` accessor defined in [Data Model].

[ERR XTDE1370] It is a non-recoverable dynamic error if the `unparsed-entity-uri` function is called when there is no context node, or when the root of the tree containing the context node is not a document node.

16.6.3 unparsed-entity-public-id

```
unparsed-entity-public-id( $entity-name as xs:string) as xs:string
```

The `unparsed-entity-public-id` function returns the public identifier of the unparsed entity whose name is given by the value of the `$entity-name` argument, in the document containing the context node. It returns the zero-length string if there is no such entity, or if the entity has no public identifier. This function maps to the `dm:unparsed-entity-public-id` accessor defined in [Data Model].

[ERR XTDE1380] It is a non-recoverable dynamic error if the `unparsed-entity-public-id` function is called when there is no context node, or when the root of the tree containing the context node is not a document node.

16.6.4 generate-id

```
generate-id() as xs:string
generate-id($node as node()?) as xs:string
```

The `generate-id` function returns a string that uniquely identifies a given node. 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 is the empty sequence, the result is the zero-length string. If the argument is omitted, it defaults to the context node.

16.6.5 system-property

```
system-property($property-name as xs:string) as xs:string
```

The `$property-name` argument MUST evaluate to a lexical QName. The lexical QName is expanded as described in *5.1 Qualified Names*.

[ERR XTDE1390] It is a non-recoverable dynamic error if the value is not a valid QName, or if there is no namespace declaration in scope for the prefix of the QName. If the processor is able to detect the error statically (for example, when the argument is supplied as a string literal), then the processor MAY optionally signal this as a static error.

The `system-property` function returns a string representing the value of the system property identified by the name. If there is no such system property, the zero-length string is 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 implementations conforming to the version of XSLT specified by this document, this is the string `"2.0"`. The value will always be a string in the lexical space of the decimal data type defined in XML Schema (see [XML Schema Part 2]). This allows the value to be converted to a number for the purpose of magnitude comparisons.
- `xsl:vendor`, a string identifying the implementer of the processor
- `xsl:vendor-url`, a string containing a URL identifying the implementer of the processor; typically this is the host page (home page) of the implementer's Web site.
- `xsl:product-name`, a string containing the name of the implementation, as defined by the implementer. This SHOULD normally remain constant from one release of the product to the next. It SHOULD also be constant across platforms in cases where the same source code is used to produce compatible products for multiple execution platforms.
- `xsl:product-version`, a string identifying the version of the implementation, as defined by the implementer. This SHOULD normally vary from one release of the product to the next, and at the discretion of the implementer it MAY also vary across different execution platforms.
- `xsl:is-schema-aware`, returns the string `"yes"` in the case of a processor that claims conformance as a schema-aware XSLT processor, or `"no"` in the case of a basic XSLT processor.
- `xsl:supports-serialization`, returns the string `"yes"` in the case of a processor that offers the serialization feature, or `"no"` otherwise.
- `xsl:supports-backwards-compatibility`, returns the string `"yes"` in the case of a processor that offers the backwards compatibility feature, or `"no"` otherwise.

Some of these properties relate to the conformance levels and features offered by the processor: these options are described in *21 Conformance*.

The actual values returned for the above properties are implementation-defined.

The set of system properties that are supported, in addition to those listed above, is also implementation-defined. Implementations MUST NOT define additional system properties in the XSLT namespace.

> **Note:**
>
> An implementation must not return the value `2.0` as the value of the `xsl:version` system property unless it is conformant to XSLT 2.0.
>
> It is recognized that vendors who are enhancing XSLT 1.0 processors may wish to release interim implementations before all the mandatory features of this specification are implemented. Since such products are not conformant to XSLT 2.0, this specification cannot define their behavior. However, implementers of such products are encouraged to return a value for the `xsl:version` system property that is intermediate between 1.0 and 2.0, and to provide the `element-available` and `function-available` functions to allow users to test which features have been fully implemented.

17 Messages

```
<!-- Category: instruction -->
<xsl:message
  select? = expression
  terminate? = { "yes" | "no" }>
  <!-- Content: sequence-constructor -->
</xsl:message>
```

The `xsl:message` instruction sends a message in an implementation-defined way. The `xsl:message` instruction causes the creation of a new document, which is typically serialized and output to an implementation-defined destination. The result of the `xsl:message` instruction is an empty sequence.

The content of the message may be specified by using either or both of the optional `select` attribute and the sequence constructor that forms the content of the `xsl:message` instruction.

If the `xsl:message` instruction contains a sequence constructor, then the sequence obtained by evaluating this sequence constructor is used to construct the content of the new document node, as described in *5.7.1 Constructing Complex Content*.

If the `xsl:message` instruction has a `select` attribute, then the value of the attribute MUST be an XPath expression. The effect of the `xsl:message` instruction is then the same as if a single `xsl:copy-of` instruction with this `select` attribute were added to the start of the sequence constructor.

If the `xsl:message` instruction has no content and no `select` attribute, then an empty message is produced.

The tree produced by the `xsl:message` instruction is not technically a final result tree. The tree has no URI and processors are not REQUIRED to make the tree accessible to applications.

Note:

In many cases, the XML document produced using `xsl:message` will consist of a document node owning a single text node. However, it may contain a more complex structure.

Note:

An implementation might implement `xsl:message` by popping up an alert box or by writing to a log file. Because the order of execution of instructions is implementation-defined, the order in which such messages appear is not predictable.

The `terminate` attribute is interpreted as an attribute value template.

If the effective value of the `terminate` attribute is `yes`, then the processor MUST terminate processing after sending the message. The default value is `no`. Note that because the order of evaluation of instructions is implementation-dependent, this gives no guarantee that any particular instruction will or will not be evaluated before processing terminates.

[ERR XTMM9000] When a transformation is terminated by use of `xsl:message terminate="yes"`, the effect is the same as when a non-recoverable dynamic error occurs during the transformation.

Example: Localizing Messages

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 select="string($messages/message[@name=$name])"/>
</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>
```

18 Extensibility and Fallback

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

[DEFINITION: An **extension instruction** is an element within a sequence constructor that is in a namespace (not the XSLT namespace) designated as an extension namespace.]

[DEFINITION: An **extension function** is a function that is available for use within an XPath expression, other than a core function defined in [Functions and Operators], an additional function defined in this XSLT specification, a constructor function named after an atomic type, or a stylesheet function defined using an `xsl:function` declaration.].

This specification does not define any mechanism for creating or binding implementations of extension instructions or extension functions, and it is not REQUIRED that implementations support any such mechanism. Such mechanisms, if they exist, are implementation-defined. 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 implementation makes particular extensions available, and to specify what happens 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 retain portability.

18.1 Extension Functions

The set of functions that can be called from a FunctionCall [XP] within an XPath expression may include one or more extension functions. The expanded-QName of an extension function always has a non-null namespace URI.

18.1.1 Testing Availability of Functions

The `function-available` function can be used with the `[xsl:]use-when` attribute (see *3.12 Conditional Element Inclusion*) to explicitly control how a stylesheet behaves if a particular extension function is not available.

```
function-available($function-name as xs:string) as xs:boolean
```

```
function-available($function-name as xs:string,
                   $arity         as xs:integer) as xs:boolean
```

A function is said to be available within an XPath expression if it is present in the in-scope functions[XP] for that expression (see *5.4.1 Initializing the Static Context*). Functions in the static context are uniquely identified by the name of the function (a QName) in combination with its arity.

The value of the `$function-name` argument MUST be a string containing a lexical QName. The lexical QName is expanded into an expanded-QName using the namespace declarations in scope for the expression. If the lexical QName is unprefixed, then the standard function namespace is used in the expanded QName.

The two-argument version of the `function-available` function returns true if and only if there is an available function whose name matches the value of the `$function-name` argument and whose arity matches the value of the `$arity` argument.

The single-argument version of the `function-available` function returns true if and only if there is at least one available function (with some arity) whose name matches the value of the `$function-name` argument.

[ERR XTDE1400] It is a non-recoverable dynamic error if the argument does not evaluate to a string that is a valid QName, or if there is no namespace declaration in scope for the prefix of the QName. If the processor is able to detect the error statically (for example, when the argument is supplied as a string literal), then the processor MAY optionally signal this as a static error.

When backwards compatible behavior is enabled, the `function-available` function returns false in respect of a function name and arity for which no implementation is available (other than the fallback error function that raises a dynamic error whenever it is called). This means that it is possible (as in XSLT 1.0) to use logic such as the following to test whether a function is available before calling it:

Example: Calling an extension function with backwards-compatibility enabled

```
<summary xsl:version="1.0">
  <xsl:choose>
    <xsl:when test="function-available('my:summary')">
      <xsl:value-of select="my:summary()"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:text>Summary not available</xsl:text>
    </xsl:otherwise>
  </xsl:choose>
</summary>
```

Note:

The fact that a function with a given name is available gives no guarantee that any particular call on the function will be successful. For example, it is not possible to determine the types of the arguments expected.

Note:

In XSLT 2.0 (without backwards compatibility enabled) a static error occurs when an XPath expression references a function that is not available. This is true even in a part of the stylesheet that uses forwards-compatible behavior. Therefore, the conditional logic to test whether a function is available before calling it should normally be written in a `use-when` attribute (see *3.12 Conditional Element Inclusion*).

Example: Stylesheet portable between XSLT 1.0 and XSLT 2.0

A stylesheet that is designed to use XSLT 2.0 facilities when they are available, but to fall back to XSLT 1.0 capabilities when not, might be written using the code:

```
<out xsl:version="2.0">
  <xsl:choose>
    <xsl:when test="function-available('matches')">
      <xsl:value-of select="matches($input, '[a-z]*')"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="string-length(
                        translate($in, 'abcdefghijklmnopqrstuvwxyz', '')) = 0"/>
    </xsl:otherwise>
  </xsl:choose>
</out>
```

Here an XSLT 2.0 processor will always take the `xsl:when` branch, while a 1.0 processor will follow the `xsl:otherwise` branch. The single-argument version of the `function-available` function is used here, because that is the only version available in XSLT 1.0. Under the rules of XSLT 1.0, the call on the `matches` function is not an error, because it is never evaluated.

Example: Stylesheet portable between XSLT 2.0 and a future version of XSLT

A stylesheet that is designed to use facilities in some future XSLT version when they are available, but to fall back to XSLT 2.0

capabilities when not, might be written using code such as the following. This hypothesizes the availability in some future version of a function `pad` which pads a string to a fixed length with spaces:

```
<xsl:value-of select="pad($input, 10)"
              use-when="function-available('pad', 2)"/>
<xsl:value-of select="concat($input, string-join(
                        for $i in 1 to 10 - string-length($input)
                                              return ' ', ''))"
              use-when="not(function-available('pad', 2))"/>
```

In this case the two-argument version of `function-available` is used, because there is no requirement for this code to run under XSLT 1.0.

18.1.2 Calling Extension Functions

If the function name used in a FunctionCall [XP] within an XPath expression identifies an extension function, then to evaluate the FunctionCall [XP], the processor will first evaluate each of the arguments in the FunctionCall [XP]. If the processor has information about the data types expected by the extension function, then it MAY perform any necessary type conversions between the XPath data types and those defined by the implementation language. If multiple extension functions are available with the same name, the processor MAY decide which one to invoke based on the number of arguments, the types of the arguments, or any other criteria. The result returned by the implementation is returned as the result of the function call, again after any necessary conversions between the data types of the implementation language and those of XPath. The details of such type conversions are outside the scope of this specification.

[ERR XTDE1420] It is a non-recoverable dynamic error if the arguments supplied to a call on an extension function do not satisfy the rules defined for that particular extension function, or if the extension function reports an error, or if the result of the extension function cannot be converted to an XPath value.

Note:

Implementations may also provide mechanisms allowing extension functions to report recoverable dynamic errors, or to execute within an environment that treats some or all of the errors listed above as recoverable.

[ERR XTDE1425] When backwards compatible behavior is enabled, it is a non-recoverable dynamic error to evaluate an extension function call if no implementation of the extension function is available.

Note:

When backwards-compatible behavior is not enabled, this is a static error [XPST0017].

Note:

There is no prohibition on calling extension functions that have side-effects (for example, an extension function that writes data to a file). However, the order of execution of XSLT instructions is not defined in this specification, so the effects of such functions are unpredictable.

Implementations are not REQUIRED to perform full validation of values returned by extension functions. It is an error for an extension function to return a string containing characters that are not permitted in XML, but the consequences of this error are implementation-defined. The implementation MAY raise an error, MAY convert the string to a string containing valid characters only, or MAY treat the invalid characters as if they were permitted characters.

Note:

The ability to execute extension functions represents a potential security weakness, since untrusted stylesheets may invoke code that has privileged access to resources on the machine where the processor executes. Implementations may therefore provide mechanisms that restrict the use of extension functions by untrusted stylesheets.

All observations in this section regarding the errors that can occur when invoking extension functions apply equally when invoking extension instructions.

18.1.3 External Objects

An implementation MAY allow an extension function to return an object that does not have any natural representation in the XDM data model, either as an atomic value or as a node. For example, an extension function `sql:connect` might return an object that represents a connection to a relational database; the resulting connection object might be passed as an argument to calls on other extension functions such as `sql:insert` and `sql:select`.

The way in which such objects are represented in the type system is implementation-defined. They might be represented by a completely new data type, or they might be mapped to existing data types such as `integer`, `string`, or `anyURI`.

18.1.4 Testing Availability of Types

The `type-available` function can be used, for example with the `[xsl:]use-when` attribute (see *3.12 Conditional Element Inclusion*), to explicitly control how a stylesheet behaves if a particular schema type is not available in the static context.

```
type-available($type-name as xs:string) as xs:boolean
```

A schema type (that is, a simple type or a complex type) is said to be available within an XPath expression if it is a type definition that is present in the in-scope schema types [XP] for that expression (see *5.4.1 Initializing the Static Context*). This includes built-in types, types imported using `xsl:import-schema`, and extension types defined by the implementation.

The value of the `$type-name` argument MUST be a string containing a lexical QName. The lexical QName is expanded into an expanded-QName using the namespace declarations in scope for the expression. If the lexical QName is unprefixed, then the default namespace is

used in the expanded QName.

The function returns true if and only if there is an available type whose name matches the value of the `$type-name` argument.

[ERR XTDE1428] It is a non-recoverable dynamic error if the argument does not evaluate to a string that is a valid QName, or if there is no namespace declaration in scope for the prefix of the QName. If the processor is able to detect the error statically (for example, when the argument is supplied as a string literal), then the processor MAY optionally signal this as a static error.

18.2 Extension Instructions

[DEFINITION: The extension instruction mechanism allows namespaces to be designated as **extension namespaces**. When a namespace is designated as an extension namespace and an element with a name from that namespace occurs in a sequence constructor, 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 sequence constructor*, user-defined data elements (see *3.6.2 User-defined Data Elements*) are not extension elements as defined here, and nothing in this section applies to them.

18.2.1 Designating an Extension Namespace

A namespace is designated as an extension namespace by using an `[xsl:]extension-element-prefixes` attribute on an element in the stylesheet (see *3.5 Standard Attributes*). The attribute MUST be in the XSLT namespace only if its parent element is *not* in the XSLT namespace. The value of the attribute is a whitespace-separated list of namespace prefixes. The namespace bound to each of the prefixes is designated as an extension namespace.

The default namespace (as declared by `xmlns`) may be designated as an extension namespace by including `#default` in the list of namespace prefixes.

[ERR XTSE1430] It is a static error if there is no namespace bound to the prefix on the element bearing the `[xsl:]extension-element-prefixes` attribute or, when `#default` is specified, if there is no default namespace.

The designation of a namespace as an extension namespace is effective for the element bearing the `[xsl:]extension-element-prefixes` attribute and for all descendants of that element within the same stylesheet module.

18.2.2 Testing Availability of Instructions

The `element-available` function can be used with the `xsl:choose` and `xsl:if` instructions, or with the `[xsl:]use-when` attribute (see *3.12 Conditional Element Inclusion*) to explicitly control how a stylesheet behaves when a particular XSLT instruction or extension instruction is (or is not) available.

```
element-available($element-name as xs:string) as xs:boolean
```

The value of the `$element-name` argument MUST be a string containing a QName. The QName is expanded into an expanded-QName using the namespace declarations in scope for the expression. If there is a default namespace in scope, then it is used to expand an unprefixed QName. The `element-available` function returns true if and only if the expanded-QName is the name of an instruction. If the expanded-QName 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 instruction. If the expanded-QName has a null namespace URI, the `element-available` function will return false.

[ERR XTDE1440] It is a non-recoverable dynamic error if the argument does not evaluate to a string that is a valid QName, or if there is no namespace declaration in scope for the prefix of the QName. If the processor is able to detect the error statically (for example, when the argument is supplied as a string literal), then the processor MAY optionally signal this as a static error.

If the expanded-QName is in the XSLT namespace, the function returns true if and only if the expanded QName is the name of an XSLT instruction, that is, an XSLT element whose syntax summary in this specification classifies it as an instruction.

> **Note:**
>
> Although the result of applying this function to a name in the XSLT namespace when using a conformant XSLT 2.0 processor is entirely predictable, the function is useful in cases where the stylesheet might be executing under a processor that implements some other version of XSLT with different rules.

If the expanded-QName is not in the XSLT namespace, the function returns true if and only if the processor has an implementation available of an extension instruction with the given expanded QName. This applies whether or not the namespace has been designated as an extension namespace.

If the processor does not have an implementation of a particular extension instruction available, and such an extension instruction is evaluated, then the processor MUST perform fallback for the element as specified in *18.2.3 Fallback*. An implementation MUST NOT signal an error merely because the stylesheet contains an extension instruction for which no implementation is available.

18.2.3 Fallback

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

The content of an `xsl:fallback` element is a sequence constructor, and when performing fallback, the value returned by the `xsl:fallback` element is the result of evaluating this sequence constructor.

When not performing fallback, evaluating an `xsl:fallback` element returns an empty sequence: the content of the `xsl:fallback` element is ignored.

There are two situations where a processor performs fallback: when an extension instruction that is not available is evaluated, and when an instruction in the XSLT namespace, that is not defined in XSLT 2.0, is evaluated within a region of the stylesheet for which forwards

compatible behavior is enabled.

Note:

Fallback processing is not invoked in other situations, for example it is not invoked when an XPath expression uses unrecognized syntax or contains a call to an unknown function. To handle such situations dynamically, the stylesheet should call functions such as `system-property` and `function-available` to decide what capabilities are available.

[ERR XTDE1450] When a processor performs fallback for an extension instruction that is not recognized, if the instruction element has one or more `xsl:fallback` children, then the content of each of the `xsl:fallback` children MUST be evaluated; it is a non-recoverable dynamic error if it has no `xsl:fallback` children.

Note:

This is different from the situation with unrecognized XSLT elements. As explained in *3.9 Forwards-Compatible Processing*, an unrecognized XSLT element appearing within a sequence constructor is a static error unless (a) forwards-compatible behavior is enabled, and (b) the instruction has an `xsl:fallback` child.

19 Final Result Trees

The output of a transformation is a set of one or more final result trees.

A final result tree can be created explicitly, by evaluating an `xsl:result-document` instruction. As explained in *2.4 Executing a Transformation*, a final result tree is also created implicitly if no `xsl:result-document` instruction is evaluated, or if the result of evaluating the initial template is a non-empty sequence.

The way in which a final result tree is delivered to an application is implementation-defined.

Serialization of final result trees is described further in *20 Serialization*

19.1 Creating Final Result Trees

```
<!-- Category: instruction -->
<xsl:result-document
  format? = { qname }
  href? = { uri-reference }
  validation? = "strict" | "lax" | "preserve" | "strip"
  type? = qname
  method? = { "xml" | "html" | "xhtml" | "text" | qname-but-not-ncname }
  byte-order-mark? = { "yes" | "no" }
  cdata-section-elements? = { qnames }
  doctype-public? = { string }
  doctype-system? = { string }
  encoding? = { string }
  escape-uri-attributes? = { "yes" | "no" }
  include-content-type? = { "yes" | "no" }
  indent? = { "yes" | "no" }
  media-type? = { string }
  normalization-form? = { "NFC" | "NFD" | "NFKC" | "NFKD" | "fully-normalized" | "none" | nmtoken }
  omit-xml-declaration? = { "yes" | "no" }
  standalone? = { "yes" | "no" | "omit" }
  undeclare-prefixes? = { "yes" | "no" }
  use-character-maps? = qnames
  output-version? = { nmtoken }>
  <!-- Content: sequence-constructor -->
</xsl:result-document>
```

The `xsl:result-document` instruction is used to create a final result tree. The content of the `xsl:result-document` element is a sequence constructor for the children of the document node of the tree. A document node is created, and the sequence obtained by evaluating the sequence constructor is used to construct the content of the document, as described in *5.7.1 Constructing Complex Content*. The tree rooted at this document node forms the final result tree.

The `xsl:result-document` instruction defines the URI of the result tree, and may optionally specify the output format to be used for serializing this tree.

The effective value of the `format` attribute, if specified, MUST be a lexical QName. The QName is expanded using the namespace declarations in scope for the `xsl:result-document` element. The expanded-QName MUST match the expanded QName of a named output definition in the stylesheet. This identifies the `xsl:output` declaration that will control the serialization of the final result tree (see *20 Serialization*), if the result tree is serialized. If the `format` attribute is omitted, the unnamed output definition is used to control serialization of the result tree.

[ERR XTDE1460] It is a non-recoverable dynamic error if the effective value of the `format` attribute is not a valid lexical QName, or if it does not match the expanded-QName of an output definition in the stylesheet. If the processor is able to detect the error statically (for example, when the `format` attribute contains no curly brackets), then the processor MAY optionally signal this as a static error.

Note:

The only way to select the unnamed output definition is to omit the `format` attribute.

The attributes `method`, `byte-order-mark` `cdata-section-elements`, `doctype-public`, `doctype-system`, `encoding`, `escape-uri-attributes`, `indent`, `media-type`, `normalization-form`, `omit-xml-declaration`, `standalone`, `undeclare-prefixes`, `use-character-maps`, and `output-version` may be used to override attributes defined in the selected output definition.

With the exception of `use-character-maps`, these attributes are all defined as attribute value templates, so their values may be set dynamically. For any of these attributes that is present on the `xsl:result-document` instruction, the effective value of the attribute overrides or supplements the corresponding value from the output definition. This works in the same way as when one `xsl:output` declaration overrides another:

- In the case of `cdata-section-elements`, the value of the serialization parameter is the union of the expanded names of the elements named in this instruction and the elements named in the selected output definition;
- In the case of `use-character-maps`, the character maps referenced in this instruction supplement and take precedence over those defined in the selected output definition;
- In all other cases, the effective value of an attribute actually present on this instruction takes precedence over the value defined in the selected output definition.

Note:

In the case of the attributes `method`, `cdata-section-elements`, and `use-character-maps`, the effective value of the attribute contains one or more lexical QNames. The prefix in such a QName is expanded using the in-scope namespaces for the `xsl:result-document` element. In the case of `cdata-section-elements`, an unprefixed element name is expanded using the default namespace.

The `output-version` attribute on the `xsl:result-document` instruction overrides the `version` attribute on `xsl:output` (it has been renamed because `version` is available with a different meaning as a standard attribute: see *3.5 Standard Attributes*). In all other cases, attributes correspond if they have the same name.

There are some serialization parameters that apply to some output methods but not to others. For example, the `indent` attribute has no effect on the `text` output method. If a value is supplied for an attribute that is inapplicable to the output method, its value is not passed to the serializer. The processor MAY validate the value of such an attribute, but is not REQUIRED to do so.

The `href` attribute is optional. The default value is the zero-length string. The effective value of the attribute MUST be a URI Reference, which may be absolute or relative. There MAY be implementation-defined restrictions on the form of absolute URI that may be used, but the implementation is not REQUIRED to enforce any restrictions. Any legal relative URI MUST be accepted. Note that the zero-length string is a legal relative URI.

The base URI of the document node at the root of the final result tree is based on the effective value of the `href` attribute. If the effective value is a relative URI, then it is resolved relative to the base output URI. If the implementation provides an API to access final result trees, then it MUST allow a final result tree to be identified by means of this base URI.

Note:

The base URI of the final result tree is not necessarily the same thing as the URI of its serialized representation on disk, if any. For example, a server (or browser client) might store final result trees only in memory, or in an internal disk cache. As long as the processor satisfies requests for those URIs, it is irrelevant where they are actually written on disk, if at all.

Note:

It will often be the case that one final result tree contains links to another final result tree produced during the same transformation, in the form of a relative URI. The mechanism of associating a URI with a final result tree has been chosen to allow the integrity of such links to be preserved when the trees are serialized.

As well as being potentially significant in any API that provides access to final result trees, the base URI of the new document node is relevant if the final result tree, rather than being serialized, is supplied as input to a further transformation.

The optional attributes `type` and `validation` may be used on the `xsl:result-document` instruction to validate the contents of the new document, and to determine the type annotation that elements and attributes within the final result tree will carry. The permitted values and their semantics are described in *19.2.2 Validating Document Nodes*.

A processor MAY allow a final result tree to be serialized. Serialization is described in *20 Serialization*. However, an implementation (for example, a processor running in an environment with no access to writable filestore) is not REQUIRED to support the serialization of final result trees. An implementation that does not support the serialization of final result trees MAY ignore the `format` attribute and the serialization attributes. Such an implementation MUST provide the application with some means of access to the (un-serialized) result tree, using its URI to identify it.

Implementations may provide additional mechanisms, outside the scope of this specification, for defining the way in which final result trees are processed. Such mechanisms MAY make use of the XSLT-defined attributes on the `xsl:result-document` and/or `xsl:output` elements, or they MAY use additional elements or attributes in an implementation-defined namespace.

Example: Multiple Result Documents

The following example takes an XHTML document as input, and breaks it up so that the text following each <h1> element is included in a separate document. A new document `toc.html` is constructed to act as an index:

```
<xsl:stylesheet
        version="2.0"
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
        xmlns:xhtml="http://www.w3.org/1999/xhtml">

<xsl:output name="toc-format" method="xhtml" indent="yes"
            doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
            doctype-public="-//W3C//DTD XHTML 1.0 Strict//EN"/>

<xsl:output name="section-format" method="xhtml" indent="no"
            doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"
            doctype-public="-//W3C//DTD XHTML 1.0 Transitional//EN"/>

<xsl:template match="/">
  <xsl:result-document href="toc.html" format="toc-format" validation="strict">
    <html xmlns="http://www.w3.org/1999/xhtml">
      <head><title>Table of Contents</title></head>
      <body>
        <h1>Table of Contents</h1>
        <xsl:for-each select="/*/xhtml:body/(*[1] | xhtml:h1)">
          <p><a href="section{position()}.html"><xsl:value-of select="."/></a></p>
        </xsl:for-each>
      </body>
```

```
    </html>
  </xsl:result-document>
  <xsl:for-each-group select="/*/xhtml:body/*" group-starting-with="xhtml:h1">
    <xsl:result-document href="section{position()}.html"
                         format="section-format" validation="strip">
      <html xmlns="http://www.w3.org/1999/xhtml">
        <head><title><xsl:value-of select="."/></title></head>
        <body>
          <xsl:copy-of select="current-group()"/>
        </body>
      </html>
    </xsl:result-document>
  </xsl:for-each-group>
</xsl:template>

</xsl:stylesheet>
```

There are restrictions on the use of the `xsl:result-document` instruction, designed to ensure that the results are fully interoperable even when processors optimize the sequence in which instructions are evaluated. Informally, the restriction is that the `xsl:result-document` instruction can only be used while writing a final result tree, not while writing to a temporary tree or a sequence. This restriction is defined formally as follows.

[DEFINITION: Each instruction in the stylesheet is evaluated in one of two possible **output states**: final output state or temporary output state].

[DEFINITION: The first of the two output states is called **final output** state. This state applies when instructions are writing to a final result tree.]

[DEFINITION: The second of the two output states is called **temporary output** state. This state applies when instructions are writing to a temporary tree or any other non-final destination.]

The instructions in the initial template are evaluated in final output state. An instruction is evaluated in the same output state as its calling instruction, except that `xsl:variable`, `xsl:param`, `xsl:with-param`, `xsl:attribute`, `xsl:comment`, `xsl:processing-instruction`, `xsl:namespace`, `xsl:value-of`, `xsl:function`, `xsl:key`, `xsl:sort`, and `xsl:message` always evaluate the instructions in their contained sequence constructor in temporary output state.

[ERR XTDE1480] It is a non-recoverable dynamic error to evaluate the `xsl:result-document` instruction in temporary output state.

[ERR XTDE1490] It is a non-recoverable dynamic error for a transformation to generate two or more final result trees with the same URI.

> **Note:**
>
> Note, this means that it is an error to evaluate more than one `xsl:result-document` instruction that omits the `href` attribute, or to evaluate any `xsl:result-document` instruction that omits the `href` attribute if an initial final result tree is created implicitly.

Technically, the result of evaluating the `xsl:result-document` instruction is an empty sequence. This means it does not contribute any nodes to the result of the sequence constructor it is part of.

[ERR XTRE1495] It is a recoverable dynamic error for a transformation to generate two or more final result trees with URIs that identify the same physical resource. The optional recovery action is implementation-dependent, since it may be impossible for the processor to detect the error.

[ERR XTRE1500] It is a recoverable dynamic error for a stylesheet to write to an external resource and read from the same resource during a single transformation, whether or not the same URI is used to access the resource in both cases. The optional recovery action is implementation-dependent: implementations are not REQUIRED to detect the error condition. Note that if the error is not detected, it is undefined whether the document that is read from the resource reflects its state before or after the result tree is written.

19.2 Validation

It is possible to control the type annotation applied to individual element and attribute nodes as they are constructed. This is done using the `type` and `validation` attributes of the `xsl:element`, `xsl:attribute`, `xsl:copy`, `xsl:copy-of`, `xsl:document`, and `xsl:result-document` instructions, or the `xsl:type` and `xsl:validation` attributes of a literal result element.

The `[xsl:]type` attribute is used to request validation of an element or attribute against a specific simple or complex type defined in a schema. The `[xsl:]validation` attribute is used to request validation against the global element or attribute declaration whose name matches the name of the element or attribute being validated.

The `[xsl:]type` and `[xsl:]validation` attributes are mutually exclusive. Both are optional, but if one is present then the other MUST be omitted. If both attributes are omitted, the effect is the same as specifying the `validation` attribute with the value specified in the `default-validation` attribute of the containing `xsl:stylesheet` element; if this is not specified, the effect is the same as specifying `validation="strip"`.

[ERR XTSE1505] It is a static error if both the `[xsl:]type` and `[xsl:]validation` attributes are present on the `xsl:element`, `xsl:attribute`, `xsl:copy`, `xsl:copy-of`, `xsl:document`, or `xsl:result-document` instructions, or on a literal result element.

The detailed rules for validation vary depending on the kind of node being validated. The rules for element and attribute nodes are given in *19.2.1 Validating Constructed Elements and Attributes*, while those for document nodes are given in *19.2.2 Validating Document Nodes*.

19.2.1 Validating Constructed Elements and Attributes

19.2.1.1 Validation using the `[xsl:]validation` Attribute

The `[xsl:]validation` attribute defines the validation action to be taken. It determines not only the type annotation of the node that is constructed by the relevant instruction itself, but also the type annotations of all element and attribute nodes that have the constructed node as an ancestor. Conceptually, the validation requested for a child element or attribute node is applied before the validation requested for its parent element. For example, if the instruction that constructs a child element specifies `validation="strict"`, this will cause the child element to be checked against an element declaration, but if the instruction that constructs its parent element specifies `validation="strip"`,

then the final effect will be that the child node is annotated as `xs:untyped`.

In the paragraphs below, the term *contained nodes* means the elements and attributes that have the newly constructed node as an ancestor.

- The value `strip` indicates that the new node and each of the contained nodes will have the type annotation `xs:untyped` if it is an element, or `xs:untypedAtomic` if it is an attribute. Any previous type annotation present on a contained element or attribute node (for example, a type annotation that is present on an element copied from a source document) is also replaced by `xs:untyped` or `xs:untypedAtomic` as appropriate. The typed value of the node is changed to be the same as its string value, as an instance of `xs:untypedAtomic`. In the case of elements the `nilled` property is set to `false`. The values of the `is-id` and `is-idrefs` properties are unchanged. Schema validation is not invoked.
- The value `preserve` indicates that nodes that are copied will retain their type annotations, but nodes whose content is newly constructed will be annotated as `xs:anyType` in the case of elements, or `xs:untypedAtomic` in the case of attributes. Schema validation is not invoked. The detailed effect depends on the instruction:
  - In the case of `xsl:element` and literal result elements, the new element has a type annotation of `xs:anyType`, and the type annotations of contained nodes are retained unchanged.
  - In the case of `xsl:attribute`, the effect is exactly the same as specifying `validation="strip"`: that is, the new attribute will have the type annotation `xs:untypedAtomic`.
  - In the case of `xsl:copy-of`, all the nodes that are copied will retain their type annotations unchanged.
  - In the case of `xsl:copy`, the effect depends on the kind of node being copied.
    1. Where the node being copied is an attribute, the copied attribute will retain its type annotation.
    2. Where the node being copied is an element, the copied element will have a type annotation of `xs:anyType` (because this instruction does not copy the content of the element, it would be wrong to assume that the type is unchanged); but any contained nodes will have their type annotations retained in the same way as with `xsl:element`.
- The value `strict` indicates that type annotations are established by performing strict schema validity assessment on the element or attribute node created by this instruction as follows:
  - In the case of an element, a top-level element declaration is identified whose local name and namespace (if any) match the name of the element, and schema-validity assessment is carried out according to the rules defined in [XML Schema Part 1] (section 3.3.4 "Element Declaration Validation Rules", validation rule "Schema-Validity Assessment (Element)", clauses 1.1 and 2, using the top-level element declaration as the "declaration stipulated by the processor", which is mentioned in clause 1.1.1.1). The element is considered valid if the result of the schema validity assessment is a PSVI in which the relevant element node has a `validity` property whose value is `valid`. If there is no matching element declaration, or if the element is not considered valid, the transformation fails [see ERR XTTE1510], [see ERR XTTE1512]. In effect this means that the element being validated MUST be declared using a top-level declaration in the schema, and MUST conform to its declaration. The process of validation applies recursively to contained elements and attributes to the extent required by the schema definition.

    Note:

    It is not an error if the identified type definition is a simple type, although [XML Schema Part 1] does not define explicitly that this case is permitted.
  - In the case of an attribute, a top-level attribute declaration is identified whose local name and namespace (if any) match the name of the attribute, and schema-validity assessment is carried out according to the rules defined in [XML Schema Part 1] (section 3.2.4 "Attribute Declaration Validation Rules", validation rule "Schema-Validity Assessment (Attribute)"). The attribute is considered valid if the result of the schema validity assessment is a PSVI in which the relevant attribute node has a `validity` property whose value is `valid`. If the attribute is not considered valid, the transformation fails [see ERR XTTE1510]. In effect this means that the attribute being validated MUST be declared using a top-level declaration in the schema, and MUST conform to its declaration.
  - The schema components used to validate an element or attribute may be located in any way described by [XML Schema Part 1] (see section 4.3.2, *How schema documents are located on the Web*). The components in the schema constructed from the synthetic schema document (see *3.14 Importing Schema Components*) will always be available for validating constructed nodes; if additional schema components are needed, they MAY be located in other ways, for example implicitly from knowledge of the namespace in which the elements and attributes appear, or using the `xsi:schemaLocation` attribute of elements within the tree being validated.
  - If no validation is performed for a node, which can happen when the schema specifies `lax` or `skip` validation for that node or for a subtree, then the node is annotated as `xs:anyType` in the case of an element, and `xs:untypedAtomic` in the case of an attribute.
- The value `lax` has the same effect as the value `strict`, except that whereas `strict` validation fails if there is no matching top-level element declaration or if the outcome of validity assessment is a `validity` property of `invalid` or `notKnown`, `lax` validation fails only if the outcome of validity assessment is a `validity` property of `invalid`. That is, `lax` validation does not cause a type error when the outcome is `notKnown`.

  In practice this means that the element or attribute being validated MUST conform to its declaration if a top-level declaration is available. If no such declaration is available, then the element or attribute is not validated, but its attributes and children are validated, again with lax validation. Any nodes whose validation outcome is a `validity` property of `notKnown` are annotated as `xs:anyType` in the case of an element, and `xs:untypedAtomic` in the case of an attribute.

  Note:

  When the parent element lacks a declaration, the XML Schema specification defines the recursive checking of children and attributes as optional. For this specification, this recursive checking is required.

  Note:

  If an element that is being validated has an `xsi:type` attribute, then the value of the `xsi:type` attribute will be taken into account when performing the validation. However, the presence of an `xsi:type` attribute will not of itself cause an element to be validated: if validation against a named type is required, as distinct from validation against a top-level element declaration, then it must be requested using the XSLT `[xsl:]type` attribute on the instruction that invokes the validation, as described in section *19.2.1.2 Validation using the [xsl:]type Attribute*

[ERR XTTE1510] If the `validation` attribute of an `xsl:element`, `xsl:attribute`, `xsl:copy`, `xsl:copy-of`, or `xsl:result-document` instruction, or the `xsl:validation` attribute of a literal result element, has the effective value `strict`, and schema validity assessment concludes that the validity of the element or attribute is invalid or unknown, a type error occurs. As with other type errors, the error MAY be signaled statically if it can be detected statically.

[ERR XTTE1512] If the `validation` attribute of an `xsl:element`, `xsl:attribute`, `xsl:copy`, `xsl:copy-of`, or `xsl:result-document` instruction, or the `xsl:validation` attribute of a literal result element, has the effective value `strict`, and there is no matching top-level declaration in the schema, then a type error occurs. As with other type errors, the error MAY be signaled statically if it can be detected statically.

[ERR XTTE1515] If the `validation` attribute of an `xsl:element`, `xsl:attribute`, `xsl:copy`, `xsl:copy-of`, or `xsl:result-document` instruction, or

the `xsl:validation` attribute of a literal result element, has the effective value `lax`, and schema validity assessment concludes that the element or attribute is invalid, a type error occurs. As with other type errors, the error MAY be signaled statically if it can be detected statically.

Note:

No mechanism is provided to validate an element or attribute against a local declaration in a schema. Such validation can usually be achieved by applying validation to a containing element for which a top-level element declaration exists.

19.2.1.2 Validation using the `[xsl:]type` Attribute

The `[xsl:]type` attribute takes as its value a `QName`. This MUST be the name of a type definition included in the in-scope schema components for the stylesheet. If the QName has no prefix, it is expanded using the default namespace established using the effective `[xsl:]xpath-default-namespace` attribute if there is one; otherwise, it is taken as being a name in no namespace.

If the `[xsl:]type` attribute is present, then the newly constructed element or attribute is validated against the type definition identified by this attribute.

- In the case of an element, schema-validity assessment is carried out according to the rules defined in [XML Schema Part 1] (section 3.3.4 "Element Declaration Validation Rules", validation rule "Schema-Validity Assessment (Element)", clauses 1.2 and 2), using this type definition as the "processor-stipulated type definition". The element is considered valid if the result of the schema validity assessment is a PSVI in which the relevant element node has a `validity` property whose value is `valid`.
- In the case of an attribute, the attribute is considered valid if (in the terminology of XML Schema) the attribute's normalized value is locally valid with respect to that type definition according to the rules for "String Valid" ([XML Schema Part 1], section 3.14.4). (Normalization here refers to the process of normalizing whitespace according to the rules of the `whiteSpace` facet for the data type).
- If the element or attribute is not considered valid, as defined above, the transformation fails [see ERR XTTE1540].

[ERR XTSE1520] It is a static error if the value of the `type` attribute of an `xsl:element`, `xsl:attribute`, `xsl:copy`, `xsl:copy-of`, `xsl:document`, or `xsl:result-document` instruction, or the `xsl:type` attribute of a literal result element, is not a valid `QName`, or if it uses a prefix that is not defined in an in-scope namespace declaration, or if the QName is not the name of a type definition included in the in-scope schema components for the stylesheet.

[ERR XTSE1530] It is a static error if the value of the `type` attribute of an `xsl:attribute` instruction refers to a complex type definition.

[ERR XTTE1540] It is a type error if an `[xsl:]type` attribute is defined for a constructed element or attribute, and the outcome of schema validity assessment against that type is that the `validity` property of that element or attribute information item is other than `valid`.

Note:

Like other type errors, this error may be signaled statically if it can be detected statically. For example, the instruction `<xsl:attribute name="dob" type="xs:date">1999-02-29</xsl:attribute>` may result in a static error being signaled. If the error is not signaled statically, it will be signaled when the instruction is evaluated.

19.2.1.3 The Validation Process

As well as checking for validity against the schema, the validity assessment process causes type annotations to be associated with element and attribute nodes. If default values for elements or attributes are defined in the schema, the validation process will where necessary create new nodes containing these default values.

Validation of an element or attribute node only takes into account constraints on the content of the element or attribute. Validation rules affecting the document as a whole are not applied. Specifically, this means:

- The validation rule "Validation Root Valid (ID/IDREF)" is not applied. This means that validation will not fail if there are non-unique ID values or dangling IDREF values in the subtree being validated.
- The validation rule "Validation Rule: Identity-constraint Satisfied" is not applied.
- There is no check that the document contains unparsed entities whose names match the values of nodes of type `xs:ENTITY` or `xs:ENTITIES`. (XSLT 2.0 provides no facility to construct unparsed entities within a tree.)
- There is no check that the document contains notations whose names match the values of nodes of type `xs:NOTATION`. (The XDM data model makes no provision for notations to be represented in the tree.)

With these caveats, validating a newly constructed element, using strict or lax validation, is equivalent to the following steps:

1. The element is serialized to textual XML form, according to the rules defined in [XSLT and XQuery Serialization] using the XML output method, with all parameters defaulted. Note that this process discards any existing type annotations.
2. The resulting XML document is parsed to create an XML Information Set (see [XML Information Set].)
3. The Information Set produced in the previous step is validated according to the rules in [XML Schema Part 1]. The result of this step is a Post-Schema Validation Infoset (PSVI). If the validation process is not successful (as defined above), a type error is raised.
4. The PSVI produced in the previous step is converted back into the XDM data model by the mapping described in [Data Model] (Section 3.3.1 Mapping PSVI Additions to Node Properties[DM]). This process creates nodes with simple or complex type annotations based on the types established during schema validation.

Validating an attribute using strict or lax validation requires a modified version of this procedure. A copy of the attribute is first added to an element node that is created for the purpose, and namespace fixup (see *5.7.3 Namespace Fixup*) is performed on this element node. The name of this element is of no consequence, but it must be the same as the name of a synthesized element declaration of the form:

```
<xs:element name="E">
  <xs:complexType>
    <xs:sequence/>
    <xs:attribute ref="A"/>
  </xs:complexType>
</xs:element>
```

where A is the name of the attribute being validated.

This synthetic element is then validated using the procedure given above for validating elements, and if it is found to be valid, a copy of the validated attribute is made, retaining its type annotation, but detaching it from the containing element (and thus, from any namespace nodes).

The XDM data model does not permit an attribute node with no parent to have a typed value that includes a namespace-qualified name, that is, a value whose type is derived from `xs:QName` or `xs:NOTATION`. This restriction is imposed because these types rely on the namespace nodes of a containing element to resolve namespace prefixes. Therefore, it is an error to validate a parentless attribute against such a type. This affects the instructions `xsl:attribute`, `xsl:copy`, and `xsl:copy-of`.

[ERR XTTE1545] A type error occurs if a `type` or `validation` attribute is defined (explicitly or implicitly) for an instruction that constructs a new attribute node, if the effect of this is to cause the attribute value to be validated against a type that is derived from, or constructed by list or union from, the primitive types `xs:QName` or `xs:NOTATION`.

19.2.2 Validating Document Nodes

It is possible to apply validation to a document node. This happens when a new document node is constructed by one of the instructions `xsl:document`, `xsl:result-document`, `xsl:copy`, or `xsl:copy-of`, and this instruction has a `type` attribute, or a `validation` attribute with the value `strict` or `lax`.

Document-level validation is not applied to the document node that is created implicitly when a variable-binding element has no `select` attribute and no `as` attribute (see *9.4 Creating implicit document nodes*). This is equivalent to using `validation="preserve"` on `xsl:document`: nodes within such trees retain their type annotation. Similarly, validation is not applied to document nodes created using `xsl:message`.

The values `validation="preserve"` and `validation="strip"` do not request validation. In the first case, all element and attribute nodes within the tree rooted at the new document node retain their type annotations. In the second case, elements within the tree have their type annotation set to `xs:untyped`, while attributes have their type annotation set to `xs:untypedAtomic`.

When validation is requested for a document node (that is, when `validation` is set to `strict` or `lax`, or when a `type` attribute is present), the following processing takes place:

- [ERR XTTE1550] A type error occurs unless the children of the document node comprise exactly one element node, no text nodes, and zero or more comment and processing instruction nodes, in any order.
- The single element node child is validated, using the supplied values of the `validation` and `type` attributes, as described in *19.2.1 Validating Constructed Elements and Attributes*.

  Note:

  The `type` attribute on `xsl:document` and `xsl:result-document`, and on `xsl:copy` and `xsl:copy-of` when copying a document node, thus refers to the required type of the element node that is the only element child of the document node. It does not refer to the type of the document node itself.
- The validation rule "Validation Root Valid (ID/IDREF)" is applied to the single element node child of the document node. This means that validation will fail if there are non-unique ID values or dangling IDREF values in the document tree.
- Identity constraints, as defined in section 3.11 of [XML Schema Part 1], are checked. (This refers to constraints defined using `xs:unique`, `xs:key`, and `xs:keyref`.)
- There is no check that the tree contains unparsed entities whose names match the values of nodes of type `xs:ENTITY` or `xs:ENTITIES`. This is because there is no facility in XSLT 2.0 to create unparsed entities in a result tree. It is possible to add unparsed entity declarations to the result document by referencing a suitable DOCTYPE during serialization.
- There is no check that the document contains notations whose names match the values of nodes of type `xs:NOTATION`. This is because notations are not part of the XDM data model. It is possible to add notations to the result document by referencing a suitable DOCTYPE during serialization.
- All other children of the document node (comments and processing instructions) are copied unchanged.

[ERR XTTE1555] It is a type error if, when validating a document node, document-level constraints are not satisfied. These constraints include identity constraints (`xs:unique`, `xs:key`, and `xs:keyref`) and ID/IDREF constraints.

20 Serialization

A processor MAY output a final result tree as a sequence of octets, although it is not REQUIRED to be able to do so (see *21 Conformance*). Stylesheet authors can use `xsl:output` declarations to specify how they wish result trees to be serialized. If a processor serializes a final result tree, it MUST do so as specified by these declarations.

The rules governing the output of the serializer are defined in [XSLT and XQuery Serialization]. The serialization is controlled using a number of serialization parameters. The values of these serialization parameters may be set within the stylesheet, using the `xsl:output`, `xsl:result-document`, and `xsl:character-map` declarations.

```
<!-- Category: declaration -->
<xsl:output
  name? = qname
  method? = "xml" | "html" | "xhtml" | "text" | qname-but-not-ncname
  byte-order-mark? = "yes" | "no"
  cdata-section-elements? = qnames
  doctype-public? = string
  doctype-system? = string
  encoding? = string
  escape-uri-attributes? = "yes" | "no"
  include-content-type? = "yes" | "no"
  indent? = "yes" | "no"
  media-type? = string
  normalization-form? = "NFC" | "NFD" | "NFKC" | "NFKD" | "fully-normalized" | "none" | nmtoken
  omit-xml-declaration? = "yes" | "no"
  standalone? = "yes" | "no" | "omit"
  undeclare-prefixes? = "yes" | "no"
  use-character-maps? = qnames
  version? = nmtoken />
```

The `xsl:output` declaration is optional; if used, it MUST always appear as a top-level element within a stylesheet module.

A stylesheet may contain multiple xsl:output declarations and may include or import stylesheet modules that also contain xsl:output declarations. The name of an xsl:output declaration is the value of its name attribute, if any.

[DEFINITION: All the xsl:output declarations in a stylesheet that share the same name are grouped into a named **output definition**; those that have no name are grouped into a single unnamed output definition.]

A stylesheet always includes an unnamed output definition; in the absence of an unnamed xsl:output declaration, the unnamed output definition is equivalent to the one that would be used if the stylesheet contained an xsl:output declaration having no attributes.

A named output definition is used when its name matches the format attribute used in an xsl:result-document element. The unnamed output definition is used when an xsl:result-document element omits the format attribute. It is also used when serializing the final result tree that is created implicitly in the absence of an xsl:result-document element.

All the xsl:output elements making up an output definition are effectively merged. For those attributes whose values are namespace-sensitive, the merging is done after lexical QNames have been converted into expanded QNames. For the cdata-section-elements attribute, the output definition uses the union of the values from all the constituent xsl:output declarations. For the use-character-maps attribute, the output definition uses the concatenation of the sequences of expanded QNames values from all the constituent xsl:output declarations, taking them in order of increasing import precedence, or where several have the same import precedence, in declaration order. For other attributes, the output definition uses the value of that attribute from the xsl:output declaration with the highest import precedence.

[ERR XTSE1560] It is a static error if two xsl:output declarations within an output definition specify explicit values for the same attribute (other than cdata-section-elements and use-character-maps), with the values of the attributes being not equal, unless there is another xsl:output declaration within the same output definition that has higher import precedence and that specifies an explicit value for the same attribute.

If none of the xsl:output declarations within an output definition specifies a value for a particular attribute, then the corresponding serialization parameter takes a default value. The default value depends on the chosen output method.

There are some serialization parameters that apply to some output methods but not to others. For example, the indent attribute has no effect on the text output method. If a value is supplied for an attribute that is inapplicable to the output method, its value is not passed to the serializer. The processor MAY validate the value of such an attribute, but is not REQUIRED to do so.

An implementation MAY allow the attributes of the xsl:output declaration to be overridden, or the default values to be changed, using the API that controls the transformation.

The location to which final result trees are serialized (whether in filestore or elsewhere) is implementation-defined (which in practice MAY mean that it is controlled using an implementation-defined API). However, these locations MUST satisfy the constraint that when two final result trees are both created (implicitly or explicitly) using relative URIs in the href attribute of the xsl:result-document instruction, then these relative URIs may be used to construct references from one tree to the other, and such references MUST remain valid when both result trees are serialized.

The method attribute on the xsl:output element identifies the overall method that is to be used for outputting the final result tree.

[ERR XTSE1570] The value MUST (if present) be a valid QName. If the QName does not have a prefix, then it identifies a method specified in [XSLT and XQuery Serialization] and MUST be one of xml, html, xhtml, or text. If the QName has a prefix, then the QName is expanded into an expanded-QName as described in *5.1 Qualified Names*; the expanded-QName identifies the output method; the behavior in this case is not specified by this document.

The default for the method attribute depends on the contents of the tree being serialized, and is chosen as follows. If the document node of the final result tree has an element child, and any text nodes preceding the first element child of the document node of the result tree contain only whitespace characters, then:

- If the expanded-QName of this first element child has local part html (in lower case), and namespace URI http://www.w3.org/1999/xhtml, then the default output method is normally xhtml. However, if the version attribute of the xsl:stylesheet element of the principal stylesheet module has the value 1.0, and if the result tree is generated implicitly (rather than by an explicit xsl:result-document instruction), then the default output method in this situation is xml.
- If the expanded-QName of this first element child has local part html (in any combination of upper and lower case) and a null namespace URI, then the default output method is html.

In all other cases, the default output method is xml.

The default output method is used if the selected output definition does not include a method attribute.

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

- The value of the encoding attribute provides the value of the encoding parameter to the serialization method. The default value is implementation-defined, but in the case of the xml and xhtml methods it MUST be either UTF-8 or UTF-16.
- The byte-order-mark attribute defines whether a byte order mark is written at the start of the file. If the value yes is specified, a byte order mark is written; if no is specified, no byte order mark is written. The default value depends on the encoding used. If the encoding is UTF-16, the default is yes; for UTF-8 it is implementation-defined, and for all other encodings it is no. The value of the byte order mark indicates whether high order bytes are written before or after low order bytes; the actual byte order used is implementation-dependent , unless it is defined by the selected encoding.
- The cdata-section-elements attribute is a whitespace-separated list of QNames. The default value is an empty list. After expansion of these names using the in-scope namespace declarations for the xsl:output declaration in which they appear, this list of names provides the value of the cdata-section-elements parameter to the serialization method. In the case of an unprefixed name, the default namespace (that is, the namespace declared using xmlns="uri") is used.

  Note:

  This differs from the rule for most other QNames used in a stylesheet. The reason is that these names refer to elements in the result document, and therefore follow the same convention as the name of a literal result element or the name attribute of xsl:element.
- The value of the doctype-system attribute provides the value of the doctype-system parameter to the serialization method. By default, the parameter is not supplied.
- The value of the doctype-public attribute provides the value of the doctype-public parameter to the serialization method. By default, the parameter is not supplied.

- The value of the `escape-uri-attributes` attribute provides the value of the `escape-uri-attributes` parameter to the serialization method. The default value is `yes`.
- The value of the `include-content-type` attribute provides the value of the `include-content-type` parameter to the serialization method. The default value is `yes`.
- The value of the `indent` attribute provides the value of the `indent` parameter to the serialization method. The default value is `yes` in the case of the `html` and `xhtml` output methods, `no` in the case of the `xml` output method.
- The value of the `media-type` attribute provides the value of the `media-type` parameter to the serialization method. The default value is `text/xml` in the case of the `xml` output method, `text/html` in the case of the `html` and `xhtml` output methods, and `text/plain` in the case of the `text` output method.
- The value of the `normalization-form` attribute provides the value of the `normalization-form` parameter to the serialization method. A value that is an `NMTOKEN` other than one of those enumerated for the `normalization-form` attribute specifes an implementation-defined normalization form; the behavior in this case is not specified by this document. The default value is `none`.
- The value of the `omit-xml-declaration` attribute provides the value of the `omit-xml-declaration` parameter to the serialization method. The default value is `no`.
- The value of the `standalone` attribute provides the value of the `standalone` parameter to the serialization method. The default value is `omit`; this means that no `standalone` attribute is to be included in the XML declaration.
- The `undeclare-prefixes` attribute is relevant only when producing output with `method="xml"` and `version="1.1"` (or later). It defines whether namespace undeclarations (of the form `xmlns:foo=""`) SHOULD be output when a child element has no namespace node with the same name (that is, namespace prefix) as a namespace node of its parent element. The default value is `no`: this means that namespace undeclarations are not output, which has the effect that when the resulting XML is reparsed, the new tree may contain namespace nodes on the child element that were not there in the original tree before serialization.
- The `use-character-maps` attribute provides a list of named character maps that are used in conjunction with this output definition. The way this attribute is used is described in *20.1 Character Maps*. The default value is an empty list.
- The value of the `version` attribute provides the value of the `version` parameter to the serialization method. The set of permitted values, and the default value, are implementation-defined. A serialization error will be reported if the requested version is not supported by the implementation.

If the processor performs serialization, then it must signal any non-recoverable serialization errors that occur. These have the same effect as non-recoverable dynamic errors: that is, the processor must signal the error and must not finish as if the transformation had been successful.

20.1 Character Maps

[DEFINITION: A **character map** allows a specific character appearing in a text or attribute node in the final result tree to be substituted by a specified string of characters during serialization.] The effect of character maps is defined in [XSLT and XQuery Serialization].

The character map that is supplied as a parameter to the serializer is determined from the `xsl:character-map` elements referenced from the `xsl:output` declaration for the selected output definition.

The `xsl:character-map` element is a declaration that may appear as a child of the `xsl:stylesheet` element.

```
<!-- Category: declaration -->
<xsl:character-map
  name = qname
  use-character-maps? = qnames>
  <!-- Content: (xsl:output-character*) -->
</xsl:character-map>
```

The `xsl:character-map` declaration declares a character map with a name and a set of character mappings. The character mappings are specified by means of `xsl:output-character` elements contained either directly within the `xsl:character-map` element, or in further character maps referenced in the `use-character-maps` attribute.

The REQUIRED `name` attribute provides a name for the character map. When a character map is used by an output definition or another character map, the character map with the highest import precedence is used.

[ERR XTSE1580] It is a static error if the stylesheet contains two or more character maps with the same name and the same import precedence, unless it also contains another character map with the same name and higher import precedence.

The optional `use-character-maps` attribute lists the names of further character maps that are included into this character map.

[ERR XTSE1590] It is a static error if a name in the `use-character-maps` attribute of the `xsl:output` or `xsl:character-map` elements does not match the `name` attribute of any `xsl:character-map` in the stylesheet.

[ERR XTSE1600] It is a static error if a character map references itself, directly or indirectly, via a name in the `use-character-maps` attribute.

It is not an error if the same character map is referenced more than once, directly or indirectly.

An output definition, after recursive expansion of character maps referenced via its `use-character-maps` attribute, may contain several mappings for the same character. In this situation, the last character mapping takes precedence. To establish the ordering, the following rules are used:

- Within a single `xsl:character-map` element, the characters defined in character maps referenced in the `use-character-maps` attribute are considered before the characters defined in the child `xsl:output-character` elements.
- The character maps referenced in a single `use-character-maps` attribute are considered in the order in which they are listed in that attribute. The expansion is depth-first: each referenced character map is fully expanded before the next one is considered.
- Two `xsl:output-character` elements appearing as children of the same `xsl:character-map` element are considered in document order.

The `xsl:output-character` element is defined as follows:

```
<xsl:output-character
  character = char
  string = string />
```

The character map that is passed as a parameter to the serializer contains a mapping for the character specified in the `character` attribute to the string specified in the `string` attribute.

Character mapping is not applied to characters for which output escaping has been disabled as described in *20.2 Disabling Output Escaping*.

If a character is mapped, then it is not subjected to XML or HTML escaping.

Example: Using Character Maps to Generate Non-XML Output

Character maps can be useful when producing serialized output in a format that resembles, but is not strictly conformant to, HTML or XML. For example, when the output is a JSP page, there might be a need to generate the output:

```
<jsp:setProperty name="user" property="id" value='<%= "id" + idValue %>'/>
```

Although this output is not well-formed XML or HTML, it is valid in Java Server Pages. This can be achieved by allocating three Unicode characters (which are not needed for any other purpose) to represent the strings `<%`, `%>`, and `"`, for example:

```
<xsl:character-map name="jsp">
  <xsl:output-character character="«" string="&lt;%"/>
  <xsl:output-character character="»" string="%&gt;"/>
  <xsl:output-character character="§" string='"'/>
</xsl:character-map>
```

When this character map is referenced in the `xsl:output` declaration, the required output can be produced by writing the following in the stylesheet:

```
<jsp:setProperty name="user" property="id" value='«= §id§ + idValue »'/>
```

This works on the assumption that when an apostrophe or quotation mark is generated as part of an attribute value by the use of character maps, the serializer will (where possible) use the other choice of delimiter around the attribute value.

Example: Constructing a Composite Character Map

The following example illustrates a composite character map constructed in a modular fashion:

```
<xsl:output name="htmlDoc" use-character-maps="htmlDoc" />

<xsl:character-map name="htmlDoc"
  use-character-maps="html-chars doc-entities windows-format" />

<xsl:character-map name="html-chars"
  use-character-maps="latin1 ..." />

<xsl:character-map name="latin1">
  <xsl:output-character character=" " string="&amp;nbsp;" />
  <xsl:output-character character="&#161;" string="&amp;iexcl;" />
  ...
</xsl:character-map>

<xsl:character-map name="doc-entities">
  <xsl:output-character character="&#xE400;" string="&amp;t-and-c;" />
  <xsl:output-character character="&#xE401;" string="&amp;chap1;" />
  <xsl:output-character character="&#xE402;" string="&amp;chap2;" />
  ...
</xsl:character-map>

<xsl:character-map name="windows-format">
  <!-- newlines as CRLF -->
  <xsl:output-character character="&#xA;" string="&#xD;&#xA;" />

  <!-- tabs as three spaces -->
  <xsl:output-character character="&#x9;" string="   " />

  <!-- images for special characters -->
  <xsl:output-character character="&#xF001;"
    string="&lt;img src='special1.gif' /&gt;" />
  <xsl:output-character character="&#xF002;"
    string="&lt;img src='special2.gif' /&gt;" />
  ...
</xsl:character-map>
```

20.2 Disabling Output Escaping

Normally, when using the XML, HTML, or XHTML output method, the serializer will escape special characters such as `&` and `<` when outputting text nodes. This ensures that the output is well-formed. 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 defines a mechanism for disabling output escaping.

This feature is deprecated.

This is an optional feature: it is not REQUIRED that a XSLT processor that implements the serialization option SHOULD offer the ability to disable output escaping, and there is no conformance level that requires this feature.

This feature requires an extension to the serializer described in [XSLT and XQuery Serialization]. Conceptually, the final result tree provides an additional boolean property `disable-escaping` associated with every character in a text node. When this property is set, the normal action of the serializer to escape special characters such as `&` and `<` is suppressed.

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 every character in the text node generated by evaluating the `xsl:value-of` or `xsl:text` element SHOULD have the `disable-output` property set.

Example: Disable Output Escaping

For example,

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

should generate the single character `<`.

If output escaping is disabled for an `xsl:value-of` or `xsl:text` instruction evaluated when temporary output state is in effect, the request to disable output escaping is ignored.

If output escaping is disabled for text within an element that would normally be output using a CDATA section, because the element is listed in the `cdata-section-elements`, then the relevant text will not be included in a CDATA section. In effect, CDATA is treated as an alternative escaping mechanism, which is disabled by the `disable-output-escaping` option.

Example: Interaction of Output Escaping and CDATA

For example, if `<xsl:output cdata-section-elements="title"/>` is specified, then the following instructions:

```
<title>
  <xsl:text disable-output-escaping="yes">This is not &lt;hr/&gt; good coding practice</xsl:text>
</title>
```

should generate the output:

```
<title><![CDATA[This is not ]]><hr/><![CDATA[ good coding practice]]></title>
```

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.

A processor will only be able to disable output escaping if it controls how the final result tree is output. This might not always be the case. For example, the result tree might be used as a source tree for another XSLT transformation instead of being output. It is implementation-defined whether (and under what circumstances) disabling output escaping is supported.

[ERR XTRE1620] It is a recoverable dynamic error if an `xsl:value-of` or `xsl:text` instruction specifies that output escaping is to be disabled and the implementation does not support this. The optional recovery action is to ignore the `disable-output-escaping` attribute.

[ERR XTRE1630] It is a recoverable dynamic error if an `xsl:value-of` or `xsl:text` instruction specifies that output escaping is to be disabled when writing to a final result tree that is not being serialized. The optional recovery action is to ignore the `disable-output-escaping` attribute.

If output escaping is disabled for a character that is not representable in the encoding that the processor is using for output, the request to disable output escaping is ignored in respect of that character.

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

Note:

The facility to define character maps for use during serialization, as described in *20.1 Character Maps*, has been produced as an alternative mechanism that can be used in many situations where disabling of output escaping was previously necessary, without the same difficulties.

21 Conformance

A processor that claims conformance with this specification MUST claim conformance either as a basic XSLT processor or as a schema-aware XSLT processor. The rules for these two conformance levels are defined in the following sections.

A processor that claims conformance at either of these two levels MAY additionally claim conformance with either or both of the following optional features: the serialization feature, defined in *21.3 Serialization Feature*, and the backwards compatibility feature, defined in *21.4 Backwards Compatibility Feature*.

Note:

There is no conformance level or feature defined in this specification that requires implementation of the static typing features described in [XPath 2.0]. An XSLT processor may provide a user option to invoke static typing, but to be conformant with this specification it must allow a stylesheet to be processed with static typing disabled. The interaction of XSLT stylesheets with the static typing feature of XPath

2.0 has not been specified, so the results of using static typing, if available, are implementation-defined.

An XSLT processor takes as its inputs a stylesheet and one or more XDM trees conforming to the data model defined in [Data Model]. It is not REQUIRED that the processor supports any particular method of constructing XDM trees, but conformance can only be tested if it provides a mechanism that enables XDM trees representing the stylesheet and primary source document to be constructed and supplied as input to the processor.

The output of the XSLT processor consists of zero or more final result trees. It is not REQUIRED that the processor supports any particular method of accessing a final result tree, but if it does not support the serialization module, conformance can only be tested if it provides some alternative mechanism that enables access to the results of the transformation.

Certain facilities in this specification are described as producing implementation-defined results. A claim that asserts conformance with this specification MUST be accompanied by documentation stating the effect of each implementation-defined feature. For convenience, a non-normative checklist of implementation-defined features is provided at *F Checklist of Implementation-Defined Features*.

A conforming processor MUST signal any static error occurring in the stylesheet, or in any XPath expression, except where specified otherwise either for individual error conditions or under the general provisions for forwards compatible behavior (see *3.9 Forwards-Compatible Processing*). After signaling such an error, the processor MAY continue for the purpose of signaling additional errors, but MUST terminate abnormally without performing any transformation.

When a dynamic error occurs during the course of a transformation, the action depends on whether the error is classified as a recoverable error. If a non-recoverable error occurs, the processor MUST signal it and MUST eventually terminate abnormally. If a recoverable error occurs, the processor MUST either signal it and terminate abnormally, or it MUST take the defined recovery action and continue processing.

Some errors, notably type errors, MAY be treated as static errors or dynamic errors at the discretion of the processor.

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

21.1 Basic XSLT Processor

[DEFINITION: A **basic XSLT processor** is an XSLT processor that implements all the mandatory requirements of this specification with the exception of certain explicitly-identified constructs related to schema processing.] These constructs are listed below.

The mandatory requirements of this specification are taken to include the mandatory requirements of XPath 2.0, as described in [XPath 2.0]. A requirement is mandatory unless the specification includes wording (such as the use of the words SHOULD or MAY) that clearly indicates that it is optional.

A basic XSLT processor MUST enforce the following restrictions. It MUST signal a static or dynamic error when the restriction is violated, as described below.

[ERR XTSE1650] A basic XSLT processor MUST signal a static error if the stylesheet includes an `xsl:import-schema` declaration.

Note:

A processor that rejects an `xsl:import-schema` declaration will also reject any reference to a user-defined type defined in a schema, or to a user-defined element or attribute declaration; it will not, however, reject references to the built-in types listed in *3.13 Built-in Types*.

[ERR XTSE1660] A basic XSLT processor MUST signal a static error if the stylesheet includes an `[xsl:]type` attribute, or an `[xsl:]validation` or `default-validation` attribute with a value other than `strip`.

A basic XSLT processor constrains the data model as follows:

- Atomic values MUST belong to one of the atomic types listed in *3.13 Built-in Types* (except as noted below).
  An atomic value may also belong to an implementation-defined type that has been added to the context for use with extension functions or extension instructions.
  The set of constructor functions available are limited to those that construct values of the above atomic types.
  The static context, which defines the full set of type names recognized by an XSLT processor and also by the XPath processor, includes these atomic types, plus `xs:anyType`, `xs:anySimpleType`, `xs:untyped`, and `xs:anyAtomicType`.
- Element nodes MUST be annotated with the type annotation `xs:untyped`, and attribute nodes with the type annotation `xs:untypedAtomic`.

[ERR XTDE1665] A basic XSLT processor MUST raise a non-recoverable dynamic error if the input to the processor includes a node with a type annotation other than `xs:untyped` or `xs:untypedAtomic`, or an atomic value of a type other than those which a basic XSLT processor supports. This error will not arise if the `input-type-annotations` attribute is set to `strip`.

Note:

Although this is expressed in terms of a requirement to detect invalid input, an alternative approach is for a basic XSLT processor to prevent this error condition occurring, by not providing any interfaces that would allow the situation to arise. A processor might, for example, implement a mapping from the PSVI to the data model that loses all non-trivial type annotations; or it might not accept input from a PSVI at all.

The phrase *input to the processor* is deliberately wide: it includes the tree containing the initial context node, trees passed as stylesheet parameters, trees accessed using the `document`, `doc` [FO], and `collection` [FO] functions, and trees returned by extension functions and extension instructions.

21.2 Schema-Aware XSLT Processor

[DEFINITION: A **schema-aware XSLT processor** is an XSLT processor that implements all the mandatory requirements of this specification, including those features that a basic XSLT processor signals as an error. The mandatory requirements of this specification are taken to include the mandatory requirements of XPath 2.0, as described in [XPath 2.0]. A requirement is mandatory unless the specification includes wording (such as the use of the words SHOULD or MAY) that clearly indicates that it is optional.]

21.3 Serialization Feature

[DEFINITION: A processor that claims conformance with the **serialization feature** MUST support the conversion of a final result tree to a sequence of octets following the rules defined in *20 Serialization*.] It MUST respect all the attributes of the `xsl:output` and `xsl:character-map` declarations, and MUST provide all four output methods, `xml`, `xhtml`, `html`, and `text`. Where the specification uses words such as MUST and REQUIRED, then it MUST serialize the result tree in precisely the way described; in other cases it MAY use an alternative, equivalent representation.

A processor may claim conformance with the serialization feature whether or not it supports the setting `disable-output-escaping="yes"` on `xsl:text`, or `xsl:value-of`.

A processor that does not claim conformance with the serialization feature MUST NOT signal an error merely because the stylesheet contains `xsl:output` or `xsl:character-map` declarations, or serialization attributes on the `xsl:result-document` instruction. Such a processor MAY check that these declarations and attributes have valid values, but is not REQUIRED to do so. Apart from optional validation, these declarations SHOULD be ignored.

21.4 Backwards Compatibility Feature

[DEFINITION: A processor that claims conformance with the **backwards compatibility feature** MUST support the processing of stylesheet instructions and XPath expressions with backwards compatible behavior, as defined in *3.8 Backwards-Compatible Processing*.]

Note that a processor that does not claim conformance with the backwards compatibility feature MUST raise a non-recoverable dynamic error if an instruction is evaluated containing an `[xsl:]version` attribute that invokes backwards compatible behavior [see ERR XTDE0160].

Note:

The reason this is a dynamic error rather than a static error is to allow stylesheets to contain conditional logic, following different paths depending on whether the XSLT processor implements XSLT 1.0 or XSLT 2.0. The selection of which path to use can be controlled by using the `system-property` function to test the `xsl:version` system property.

A processor that claims conformance with the backwards compatibility feature MUST permit the use of the namespace axis in XPath expressions when backwards compatible behavior is enabled. In all other circumstances, support for the namespace axis is optional.

A References

A.1 Normative References

Data Model
XQuery 1.0 and XPath 2.0 Data Model (XDM), Norman Walsh, Mary Fernández, Ashok Malhotra, *et. al.*, Editors. World Wide Web Consortium, 23 Jan 2007. This version is http://www.w3.org/TR/2007/REC-xpath-datamodel-20070123/. The latest version is available at http://www.w3.org/TR/xpath-datamodel/.

Functions and Operators
XQuery 1.0 and XPath 2.0 Functions and Operators, Ashok Malhotra, Jim Melton, and Norman Walsh, Editors. World Wide Web Consortium, 23 Jan 2007. This version is http://www.w3.org/TR/2007/REC-xpath-functions-20070123/. The latest version is available at http://www.w3.org/TR/xpath-functions/.

XML Information Set
XML Information Set (Second Edition), John Cowan and Richard Tobin, Editors. World Wide Web Consortium, 04 Feb 2004. This version is http://www.w3.org/TR/2004/REC-xml-infoset-20040204. The latest version is available at http://www.w3.org/TR/xml-infoset.

ISO 3166-1
ISO (International Organization for Standardization) *Codes for the representation of names of countries and their subdivisions - Part 1: Country codes* ISO 3166-1:1997.

ISO 8601
ISO (International Organization for Standardization) *Data elements and interchange formats - Information interchange - Representation of dates and times.* ISO 8601:2000(E), Second edition, 2000-12-15.

XSLT and XQuery Serialization
XSLT 2.0 and XQuery 1.0 Serialization, Joanne Tong, Michael Kay, Norman Walsh, *et. al.*, Editors. World Wide Web Consortium, 23 Jan 2007. This version is http://www.w3.org/TR/2007/REC-xslt-xquery-serialization-20070123/. The latest version is available at http://www.w3.org/TR/xslt-xquery-serialization/.

XML 1.0
Extensible Markup Language (XML) 1.0 (Fourth Edition), Eve Maler, Jean Paoli, François Yergeau, *et. al.*, Editors. World Wide Web Consortium, 16 Aug 2006. This version is http://www.w3.org/TR/2006/REC-xml-20060816. The latest version is available at http://www.w3.org/TR/xml.

XML 1.1
Extensible Markup Language (XML) 1.1 (Second Edition), Tim Bray, John Cowan, Jean Paoli, *et. al.*, Editors. World Wide Web Consortium, 16 Aug 2006. This version is http://www.w3.org/TR/2006/REC-xml11-20060816. The latest version is available at http://www.w3.org/TR/xml11/.

XML Base
XML Base, Jonathan Marsh, Editor. World Wide Web Consortium, 27 Jun 2001. This version is http://www.w3.org/TR/2001/REC-xmlbase-20010627/. The latest version is available at http://www.w3.org/TR/xmlbase/.

xml:id
xml:id Version 1.0, Norman Walsh, Daniel Veillard, and Jonathan Marsh, Editors. World Wide Web Consortium, 09 Sep 2005. This version is http://www.w3.org/TR/2005/REC-xml-id-20050909/. The latest version is available at http://www.w3.org/TR/xml-id/.

Namespaces in XML 1.0
Namespaces in XML, Andrew Layman, Dave Hollander, and Tim Bray, Editors. World Wide Web Consortium, 14 Jan 1999. This version is http://www.w3.org/TR/1999/REC-xml-names-19990114. The latest version is available at http://www.w3.org/TR/REC-xml-names.

Namespaces in XML 1.1
Namespaces in XML 1.1 (Second Edition), Tim Bray, Dave Hollander, Andrew Layman, and Richard Tobin, Editors. World Wide Web Consortium, 16 Aug 2006. This version is http://www.w3.org/TR/2006/REC-xml-names11-20060816. The latest version is available at http://www.w3.org/TR/xml-names11/.

XML Schema Part 1
XML Schema Part 1: Structures Second Edition, Henry S. Thompson, David Beech, Noah Mendelsohn, and Murray Maloney, Editors. World Wide Web Consortium, 28 Oct 2004. This version is http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/. The latest version is available at http://www.w3.org/TR/xmlschema-1/.

XML Schema Part 2

XML Schema Part 2: Datatypes Second Edition, Paul V. Biron and Ashok Malhotra, Editors. World Wide Web Consortium, 28 Oct 2004. This version is http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/. The latest version is available at http://www.w3.org/TR/xmlschema-2/.

XPath 2.0
XML Path Language (XPath) 2.0, Don Chamberlin , Anders Berglund, Scott Boag, *et. al.*, Editors. World Wide Web Consortium, 23 Jan 2007. This version is http://www.w3.org/TR/2007/REC-xpath20-20070123/. The latest version is available at http://www.w3.org/TR/xpath20/.

A.2 Other References

Calendrical Calculations
Edward M. Reingold and Nachum Dershowitz. *Calendrical Calculations Millennium edition (2nd Edition)*. Cambridge University Press, ISBN 0 521 77752 6

DOM Level 2
Document Object Model (DOM) Level 2 Core Specification, Philippe Le Hégaret, Steve Byrne, Arnaud Le Hors, *et. al.*, Editors. World Wide Web Consortium, 13 Nov 2000. This version is http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113. The latest version is available at http://www.w3.org/TR/DOM-Level-2-Core/.

RFC2119
S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels*. IETF RFC 2119. See http://www.ietf.org/rfc/rfc2119.txt.

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

RFC3023
M. Murata, S. St.Laurent, and D. Cohn. *XML Media Types*. IETF RFC 3023. See http://www.ietf.org/rfc/rfc3023.txt. References to RFC 3023 should be taken to refer to any document that supersedes RFC 3023.

RFC3986
T. Berners-Lee, R. Fielding, and L. Masinter. *Uniform Resource Identifiers (URI): Generic Syntax*. IETF RFC 3986. See http://www.ietf.org/rfc/rfc3986.txt.

RFC3987
M. Duerst, M. Suignard. *Internationalized Resource Identifiers (IRIs)*. IETF RFC 3987. See http://www.ietf.org/rfc/rfc3987.txt.

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

XInclude
XML Inclusions (XInclude) Version 1.0 (Second Edition), David Orchard, Jonathan Marsh, and Daniel Veillard, Editors. World Wide Web Consortium, 15 Nov 2006. This version is http://www.w3.org/TR/2006/REC-xinclude-20061115/. The latest version is available at http://www.w3.org/TR/xinclude/.

XLink
XML Linking Language (XLink) Version 1.0, David Orchard, Eve Maler, and Steven DeRose, Editors. World Wide Web Consortium, 27 Jun 2001. This version is http://www.w3.org/TR/2001/REC-xlink-20010627/. The latest version is available at http://www.w3.org/TR/xlink/.

XML Schema 1.0 and XML 1.1
World Wide Web Consortium. *Processing XML 1.1 documents with XML Schema 1.0 processors*. W3C Working Group Note 11 May 2005. See http://www.w3.org/TR/2005/NOTE-xml11schema10-20050511/

XML Stylesheet
Associating Style Sheets with XML documents, James Clark, Editor. World Wide Web Consortium, 29 Jun 1999. This version is http://www.w3.org/1999/06/REC-xml-stylesheet-19990629. The latest version is available at http://www.w3.org/TR/xml-stylesheet.

XPointer Framework
XPointer Framework, Paul Grosso, Jonathan Marsh, Eve Maler, and Norman Walsh, Editors. World Wide Web Consortium, 25 Mar 2003. This version is http://www.w3.org/TR/2003/REC-xptr-framework-20030325/. The latest version is available at http://www.w3.org/TR/xptr-framework/.

Extensible Stylesheet Language (XSL)
Extensible Stylesheet Language (XSL) Version 1.1, Anders Berglund, Editor. World Wide Web Consortium, 05 Dec 2006. This version is http://www.w3.org/TR/2006/REC-xsl11-20061205/. The latest version is available at http://www.w3.org/TR/xsl/.

XSLT 1.0
XSL Transformations (XSLT) Version 1.0, James Clark, Editor. World Wide Web Consortium, 16 Nov 1999. This version is http://www.w3.org/TR/1999/REC-xslt-19991116. The latest version is available at http://www.w3.org/TR/xslt.

XSLT 2.0 Requirements
XSLT Requirements Version 2.0, Steve Muench and Mark Scardina, Editors. World Wide Web Consortium, 14 Feb 2001. This version is http://www.w3.org/TR/2001/WD-xslt20req-20010214. The latest version is available at http://www.w3.org/TR/xslt20req.

B The XSLT Media Type

This appendix registers a new MIME media type, "`application/xslt+xml`".

This information is being submitted to the IESG (Internet Engineering Steering Group) for review, approval, and registration with IANA (the Internet Assigned Numbers Authority).

B.1 Registration of MIME Media Type application/xslt+xml

MIME media type name:

`application`

MIME subtype name:

`xslt+xml`

Required parameters:

None.

Optional parameters:

`charset`

This parameter has identical semantics to the `charset` parameter of the `application/xml` media type as specified in [RFC3023].

Encoding considerations:

By virtue of XSLT content being XML, it has the same considerations when sent as "`application/xslt+xml`" as does XML. See RFC 3023, section 3.2.

Security considerations:

Several XSLT instructions may cause arbitrary URIs to be dereferenced. In this case, the security issues of [RFC3986], section 7, should be considered.

In addition, because of the extensibility features for XSLT, it is possible that "`application/xslt+xml`" may describe content that has security implications beyond those described here. However, if the processor follows only the normative semantics of this specification, this content will be ignored. Only in the case where the processor recognizes and processes the additional content, or where further processing of that content is dispatched to other processors, would security issues potentially arise. And in that case, they would fall outside the domain of this registration document.

Interoperability considerations:

This specification describes processing semantics that dictate behavior that must be followed when dealing with, among other things, unrecognized elements.

Because XSLT is extensible, conformant "`application/xslt+xml`" processors can expect that content received is well-formed XML, but it cannot be guaranteed that the content is valid XSLT or that the processor will recognize all of the elements and attributes in the document.

Published specification:

This media type registration is for XSLT stylesheet modules as described by the XSLT 2.0 specification, which is located at http://www.w3.org/TR/xslt20/. It is also appropriate to use this media type with earlier and later versions of the XSLT language.

Applications which use this media type:

Existing XSLT 1.0 stylesheets are most often described using the unregistered media type "`text/xsl`".

There is no experimental, vendor specific, or personal tree predecessor to "`application/xslt+xml`", reflecting the fact that no applications currently recognize it. This new type is being registered in order to allow for the expected deployment of XSLT 2.0 on the World Wide Web, as a first class XML application.

Additional information:

Magic number(s):

There is no single initial octet sequence that is always present in XSLT documents.

File extension(s):

XSLT documents are most often identified with the extensions "`.xsl`" or "`.xslt`".

Macintosh File Type Code(s):

TEXT

Person & email address to contact for further information:

Norman Walsh, <Norman.Walsh@Sun.COM>.

Intended usage:

COMMON

Author/Change controller:

The XSLT specification is a work product of the World Wide Web Consortium's XSL Working Group. The W3C has change control over these specifications.

B.2 Fragment Identifiers

For documents labeled as "`application/xslt+xml`", the fragment identifier notation is exactly that for "`application/xml`", as specified in RFC 3023.

C Glossary (Non-Normative)

QName

A **QName** is always written in the form `(NCName ":")? NCName`, that is, a local name optionally preceded by a namespace prefix. When two QNames are compared, however, they are considered equal if the corresponding expanded-QNames are the same, as described below.

URI Reference

Within this specification, the term **URI Reference**, unless otherwise stated, refers to a string in the lexical space of the `xs:anyURI` data type as defined in [XML Schema Part 2].

XML namespace

The **XML namespace**, defined in [Namespaces in XML 1.0] as `http://www.w3.org/XML/1998/namespace`, is used for attributes such as `xml:lang`, `xml:space`, and `xml:id`.

XPath 1.0 compatibility mode

The term **XPath 1.0 compatibility mode** is defined in Section 2.1.1 Static Context[XP]. This is a setting in the static context of an XPath expression; it has two values, `true` and `false`. When the value is set to true, the semantics of function calls and certain other operations are adjusted to give a greater degree of backwards compatibility between XPath 2.0 and XPath 1.0.

XSLT element

An **XSLT element** is an element in the XSLT namespace whose syntax and semantics are defined in this specification.

XSLT instruction

An **XSLT instruction** is an XSLT element whose syntax summary in this specification contains the annotation `<!-- category: instruction -->`.

XSLT namespace

The **XSLT namespace** has the URI `http://www.w3.org/1999/XSL/Transform`. It is used to identify elements, attributes, and other names that have a special meaning defined in this specification.

alias

A stylesheet can use the `xsl:namespace-alias` element to declare that a literal namespace URI is being used as an **alias** for a target namespace URI.

arity

The **arity** of a stylesheet function is the number of `xsl:param` elements in the function definition.

atomize

The term **atomization** is defined in Section 2.4.2 Atomization[XP]. It is a process that takes as input a sequence of nodes and atomic values, and returns a sequence of atomic values, in which the nodes are replaced by their typed values as defined in [Data Model].

attribute set

The `xsl:attribute-set` element defines a named **attribute set**: that is, a collection of attribute definitions that can be used repeatedly on different constructed elements.

attribute value template

In an attribute that is designated 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 brackets (`{}`)

backwards compatibility feature

A processor that claims conformance with the **backwards compatibility feature** MUST support the processing of stylesheet instructions and XPath expressions with backwards compatible behavior, as defined in *3.8 Backwards-Compatible Processing*.

backwards compatible behavior

An element enables backwards-compatible behavior for itself, its attributes, its descendants and their attributes if it has an `[xsl:]version` attribute (see *3.5 Standard Attributes*) whose value is less than `2.0`.

base output URI

The **base output URI** is a URI to be used as the base URI when resolving a relative URI allocated to a final result tree. If the transformation generates more than one final result tree, then typically each one will be allocated a URI relative to this base URI.

basic XSLT processor

A **basic XSLT processor** is an XSLT processor that implements all the mandatory requirements of this specification with the exception of certain explicitly-identified constructs related to schema processing.

character map

A **character map** allows a specific character appearing in a text or attribute node in the final result tree to be substituted by a specified string of characters during serialization.

circularity

A **circularity** is said to exist if a construct such as a global variable, an attribute set, or a key is defined in terms of itself. For example, if the expression or sequence constructor 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*. A circularity exists if it is impossible to do this for all global variable definitions.

collation

Facilities in XSLT 2.0 and XPath 2.0 that require strings to be ordered rely on the concept of a named **collation**. A collation is a set of rules that determine whether two strings are equal, and if not, which of them is to be sorted before the other.

context item

The **context item** is the item currently being processed. An item (see [Data Model]) is either an atomic value (such as an integer, date, or string), or a node. The context item is initially set to the initial context node supplied when the transformation is invoked (see *2.3 Initiating a Transformation*). It changes whenever instructions such as `xsl:apply-templates` and `xsl:for-each` are used to process a sequence of items; each item in such a sequence becomes the context item while that item is being processed.

context node

If the context item is a node (as distinct from an atomic value such as an integer), then it is also referred to as the **context node**. The context node is not an independent variable, it changes whenever the context item changes. When the context item is an atomic value, there is no context node.

context position

The **context position** is the position of the context item within the sequence of items currently being processed. It changes whenever the context item changes. When an instruction such as `xsl:apply-templates` or `xsl:for-each` is used to process a sequence of items, the first item in the sequence is processed with a context position of 1, the second item with a context position of 2, and so on.

context size

The **context size** is the number of items in the sequence of items currently being processed. It changes whenever instructions such as `xsl:apply-templates` and `xsl:for-each` are used to process a sequence of items; during the processing of each one of those items, the context size is set to the count of the number of items in the sequence (or equivalently, the position of the last item in the sequence).

core function

The term **core function** means a function that is specified in [Functions and Operators] and that is in the standard function namespace.

current captured substrings

While the `xsl:matching-substring` instruction is active, a set of **current captured substrings** is available, corresponding to the parenthesized sub-expressions of the regular expression.

current group

The evaluation context for XPath expressions includes a component called the **current group**, which is a sequence. The current group is the collection of related items that are processed collectively in one iteration of the `xsl:for-each-group` element.

current grouping key

The evaluation context for XPath expressions includes a component called the **current grouping key**, which is an atomic value. The current grouping key is the grouping key shared in common by all the items within the current group.

current mode

At any point in the processing of a stylesheet, there is a **current mode**. When the transformation is initiated, the current mode is the default mode, unless a different initial mode has been supplied, as described in *2.3 Initiating a Transformation*. Whenever an `xsl:apply-templates` instruction is evaluated, the current mode becomes the mode selected by this instruction.

current template rule

At any point in the processing of a stylesheet, there may be a **current template rule**. Whenever a template rule is chosen as a result of evaluating `xsl:apply-templates`, `xsl:apply-imports`, or `xsl:next-match`, the template rule becomes the current template rule for the evaluation of the rule's sequence constructor. When an `xsl:for-each`, `xsl:for-each-group`, or `xsl:analyze-string` instruction is evaluated, or when evaluating a sequence constructor contained in an `xsl:sort` or `xsl:key` element, or when a stylesheet function is called (see *10.3 Stylesheet Functions*), the current template rule becomes null for the evaluation of that instruction or function.

date formatting function

The three functions `format-date`, `format-time`, and `format-dateTime` are referred to collectively as the **date formatting functions**.

decimal format

All the `xsl:decimal-format` declarations in a stylesheet that share the same name are grouped into a named **decimal format**; those that have no name are grouped into a single unnamed decimal format.

declaration

Top-level elements fall into two categories: declarations, and user-defined data elements. Top-level elements whose names are in the XSLT namespace are **declarations**. Top-level elements in any other namespace are user-defined data elements (see *3.6.2 User-defined Data Elements*)

declaration order

The declarations within a stylesheet level have a total ordering known as **declaration order**. The order of declarations within a stylesheet level is the same as the document order that would result if each stylesheet module were inserted textually in place of the `xsl:include` element that references it.

default collation

In this specification the term **default collation** means the collation that is used by XPath operators such as `eq` and `lt` appearing in XPath expressions within the stylesheet.

default mode

There is always a **default mode** available. The default mode is an unnamed mode, and it is used when no `mode` attribute is specified on an `xsl:apply-templates` instruction.

default priority

If no `priority` attribute is specified on the `xsl:template` element, a **default priority** is computed, based on the syntax of the pattern supplied in the `match` attribute.

defining element

A string in the form of a lexical QName may occur as the value of an attribute node in a stylesheet module, or within an XPath expression contained in such an attribute node, or as the result of evaluating an XPath expression contained in such an attribute node. The element containing this attribute node is referred to as the **defining element** of the QName.

deprecated

Some constructs defined in this specification are described as being **deprecated**. The use of this term implies that stylesheet authors SHOULD NOT use the construct, and that the construct may be removed in a later version of this specification.

dynamic error

An error that is not detected until a source document is being transformed is referred to as a **dynamic error**.

effective value

The result of evaluating an attribute value template is referred to as the **effective value** of the attribute.

embedded stylesheet module

An **embedded stylesheet module** is a stylesheet module that is embedded within another XML document, typically the source document that is being transformed.

expanded-QName

An **expanded-QName** contains a pair of values, namely a local name and an optional namespace URI. It may also contain a namespace prefix. Two expanded-QNames are equal if the namespace URIs are the same (or both absent) and the local names are the same. The prefix plays no part in the comparison, but is used only if the expanded-QName needs to be converted back to a string.

expression

Within this specification, the term **XPath expression**, or simply **expression**, means a string that matches the production Expr [XP] defined in [XPath 2.0].

extension attribute

An element from the XSLT namespace may have any attribute not from the XSLT namespace, provided that the expanded-QName (see [XPath 2.0]) of the attribute has a non-null namespace URI. These attributes are referred to as **extension attributes**.

extension function

An **extension function** is a function that is available for use within an XPath expression, other than a core function defined in [Functions and Operators], an additional function defined in this XSLT specification, a constructor function named after an atomic type, or a stylesheet function defined using an `xsl:function` declaration.

extension instruction

An **extension instruction** is an element within a sequence constructor that is in a namespace (not the XSLT namespace) designated as an extension namespace.

extension namespace

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

final output state

The first of the two output states is called **final output** state. This state applies when instructions are writing to a final result tree.

final result tree

A **final result tree** is a result tree that forms part of the final output of a transformation. Once created, the contents of a final result tree are not accessible within the stylesheet itself.

focus

When a sequence constructor is evaluated, the processor keeps track of which items are being processed by means of a set of implicit variables referred to collectively as the **focus**.

forwards-compatible behavior

An element enables **forwards-compatible behavior** for itself, its attributes, its descendants and their attributes if it has an `[xsl:]version` attribute (see *3.5 Standard Attributes*) whose value is greater than `2.0`.

function conversion rules

Except where otherwise indicated, the actual value of an expression is converted to the required type using the **function conversion rules**. These are the rules defined in [XPath 2.0] for converting the supplied argument of a function call to the required type of that argument, as defined in the function signature. The relevant rules are those that apply when XPath 1.0 compatibility mode is set to `false`.

function parameter

An `xsl:param` element may appear as a child of an `xsl:function` element, before any non-`xsl:param` children of that element. Such a parameter is known as a **function parameter**. A function parameter is a local variable with the additional property that its value can be set when the function is called, using a function call in an XPath expression.

global variable

A top-level variable-binding element declares a **global variable** that is visible everywhere (except where it is shadowed by another binding).

group

The `xsl:for-each-group` instruction allocates the items in an input sequence into **groups** of items (that is, it establishes a collection of sequences) based either on common values of a grouping key, or on a pattern that the initial or final node in a group must match.

grouping key

If either of the `group-by` attribute or `group-adjacent` attributes is present, then **grouping keys** are calculated for each item in the population. The grouping keys are the items in the sequence obtained by evaluating the expression contained in the `group-by` attribute or `group-adjacent` attribute, atomizing the result, and then casting an `xs:untypedAtomic` value to `xs:string`.

implementation

A specific product that performs the functions of an XSLT processor is referred to as an **implementation**

implementation-defined

In this specification, the term **implementation-defined** refers to a feature where the implementation is allowed some flexibility, and where the choices made by the implementation MUST be described in documentation that accompanies any conformance claim.

implementation-dependent

The term **implementation-dependent** refers to a feature where the behavior MAY vary from one implementation to another, and where the vendor is not expected to provide a full specification of the behavior.

import precedence

A declaration *D* in the stylesheet is defined to have lower **import precedence** than another declaration *E* if the stylesheet level containing *D* would be visited before the stylesheet level containing *E* in a post-order traversal of the import tree (that is, a traversal of the import tree in which a stylesheet level is visited after its children). Two declarations within the same stylesheet level have the same import precedence.

import tree

The stylesheet levels making up a stylesheet are treated as forming an **import tree**. In the import tree, each stylesheet level has one child for each `xsl:import` declaration that it contains.

in-scope schema component

The schema components that may be referenced by name in a stylesheet are referred to as the **in-scope schema components**. This set is the same throughout all the modules of a stylesheet.

initial context node

A node that acts as the **initial context node** for the transformation. This node is accessible within the stylesheet as the initial value of the XPath expressions . (dot) and `self::node()`, as described in *5.4.3.1 Maintaining Position: the Focus*

initial item

For each group, the item within the group that is first in population order is known as the **initial item** of the group.

initial sequence

The sequence to be sorted is referred to as the **initial sequence**.

initial template

The transformation is performed by evaluating an **initial template**. If a named template is supplied when the transformation is initiated, then this is the initial template; otherwise, the initial template is the template rule selected according to the rules of the `xsl:apply-templates` instruction for processing the initial context node in the initial mode.

instruction

An **instruction** is either an XSLT instruction or an extension instruction.

key

A **key** is defined as a set of `xsl:key` declarations in the stylesheet that share the same name.

key specifier

The expression in the `use` attribute and the sequence constructor within an `xsl:key` declaration are referred to collectively as the **key specifier**. The key specifier determines the values that may be used to find a node using this key.

lexical QName

A **lexical QName** is a string representing a QName in the form `(NCName ":")? NCName`, that is, a local name optionally preceded by a namespace prefix.

literal namespace URI

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**.

literal result element

In a sequence constructor, an element in the stylesheet that does not belong to the XSLT namespace and that is not an extension instruction (see *18.2 Extension Instructions*) is classified as a **literal result element**.

local variable

As well as being allowed as declaration elements, the `xsl:variable` element is also allowed in sequence constructors. Such a variable is known as a **local variable**.

mode

Modes allow a node in a source tree to be processed multiple times, each time producing a different result. They also allow different sets of template rules to be active when processing different trees, for example when processing documents loaded using the `document` function (see *16.1 Multiple Source Documents*) or when processing temporary trees.

named template

Templates can be invoked by name. An `xsl:template` element with a `name` attribute defines a **named template**.

namespace fixup

The rules for the individual XSLT instructions that construct a result tree (see *11 Creating Nodes and Sequences*) prescribe some of the situations in which namespace nodes are written to the tree. These rules, however, are not sufficient to ensure that the prescribed constraints are always satisfied. The XSLT processor MUST therefore add additional namespace nodes to satisfy these constraints. This process is referred to as **namespace fixup**.

non-recoverable dynamic error

A dynamic error that is not recoverable is referred to as a **non-recoverable dynamic error**. When a non-recoverable dynamic error occurs, the processor MUST signal the error, and the transformation fails.

optional recovery action

If an implementation chooses to recover from a recoverable dynamic error, it MUST take the **optional recovery action** defined for that error condition in this specification.

order of first appearance

There is an ordering among groups referred to as the **order of first appearance**. A group *G* is defined to precede a group *H* in order of first appearance if the initial item of *G* precedes the initial item of *H* in population order. If two groups *G* and *H* have the same initial item (because the item is in both groups) then *G* precedes *H* if the grouping key of *G* precedes the grouping key of *H* in the sequence that results from evaluating the `group-by` expression of this initial item.

output definition

All the `xsl:output` declarations in a stylesheet that share the same name are grouped into a named **output definition**; those that have no name are grouped into a single unnamed output definition.

output state

Each instruction in the stylesheet is evaluated in one of two possible **output states**: final output state or temporary output state

parameter

The `xsl:param` element declares a **parameter**, which may be a stylesheet parameter, a template parameter, or a function parameter. A parameter is a variable with the additional property that its value can be set by the caller when the stylesheet, the template, or the function is invoked.

pattern

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.

picture string

The formatting of a number is controlled by a **picture string**. The picture string is a sequence of characters, in which the characters assigned to the variables *decimal-separator-sign*, *grouping-sign*, *zero-digit-sign*, *digit-sign* and *pattern-separator-sign* are classified as active characters, and all other characters (including the *percent-sign* and *per-mille-sign*) are classified as passive characters.

place marker

The `xsl:number` instruction performs two tasks: firstly, determining a **place marker** (this is a sequence of integers, to allow for hierarchic numbering schemes such as `1.12.2` or `3(c)ii`), and secondly, formatting the place marker for output as a text node in the result sequence.

population

The sequence of items to be grouped, which is referred to as the **population**, is determined by evaluating the XPath expression contained in the `select` attribute.

population order

The population is treated as a sequence; the order of items in this sequence is referred to as **population order**

principal stylesheet module

A stylesheet may consist of several stylesheet modules, contained in different XML documents. For a given transformation, one of these functions as the **principal stylesheet module**. The complete stylesheet is assembled by finding the stylesheet modules referenced directly or indirectly from the principal stylesheet module using `xsl:include` and `xsl:import` elements: see *3.10.2 Stylesheet Inclusion* and *3.10.3 Stylesheet Import*.

processing order

There is another ordering among groups referred to as **processing order**. If group *R* precedes group *S* in processing order, then in the result sequence returned by the `xsl:for-each-group` instruction the items generated by processing group *R* will precede the items generated by processing group *S*.

processor

The software responsible for transforming source trees into result trees using an XSLT stylesheet is referred to as the **processor**. This is sometimes expanded to *XSLT processor* to avoid any confusion with other processors, for example an XML processor.

recoverable error

Some dynamic errors are classed as **recoverable errors**. When a recoverable error occurs, this specification allows the processor either to signal the error (by reporting the error condition and terminating execution) or to take a defined recovery action and continue processing.

required type

The context within a stylesheet where an XPath expression appears may specify the **required type** of the expression. The required type indicates the type of the value that the expression is expected to return.

reserved namespace

The XSLT namespace, together with certain other namespaces recognized by an XSLT processor, are classified as **reserved namespaces** and MUST be used only as specified in this and related specifications.

result tree

The term **result tree** is used to refer to any tree constructed by instructions in the stylesheet. A result tree is either a final result tree or a temporary tree.

schema component

Type definitions and element and attribute declarations are referred to collectively as **schema components**.

schema instance namespace

The **schema instance namespace** `http://www.w3.org/2001/XMLSchema-instance` is used as defined in [XML Schema Part 1]

schema namespace

The **schema namespace** `http://www.w3.org/2001/XMLSchema` is used as defined in [XML Schema Part 1]

schema-aware XSLT processor

A **schema-aware XSLT processor** is an XSLT processor that implements all the mandatory requirements of this specification, including those features that a basic XSLT processor signals as an error. The mandatory requirements of this specification are taken to include the mandatory requirements of XPath 2.0, as described in [XPath 2.0]. A requirement is mandatory unless the specification includes wording (such as the use of the words SHOULD or MAY) that clearly indicates that it is optional.

sequence constructor

A **sequence constructor** is a sequence of zero or more sibling nodes in the stylesheet that can be evaluated to return a sequence of nodes and atomic values. The way that the resulting sequence is used depends on the containing instruction.

serialization

A frequent requirement is to output a final result tree as an XML document (or in other formats such as HTML). This process is referred to as **serialization**.

serialization error

If a transformation has successfully produced a final result tree, it is still possible that errors may occur in serializing the result tree. For example, it may be impossible to serialize the result tree using the encoding selected by the user. Such an error is referred to as a **serialization error**.

serialization feature

A processor that claims conformance with the **serialization feature** MUST support the conversion of a final result tree to a sequence of octets following the rules defined in *20 Serialization*.

shadows

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.

simplified stylesheet module

A **simplified stylesheet module** is a tree, or part of a tree, consisting of a literal result element together with its descendant nodes and associated attributes and namespaces. This element is not itself in the XSLT namespace, but it MUST have an `xsl:version` attribute, which implies that it MUST have a namespace node that declares a binding for the XSLT namespace. For further details see *3.7 Simplified Stylesheet Modules*.

singleton focus

A **singleton focus** based on a node *N* has the context item (and therefore the context node) set to *N*, and the context position and context size both set to 1 (one).

sort key component

Within a sort key specification, each `xsl:sort` element defines one **sort key component**.

sort key specification

A **sort key specification** is a sequence of one or more adjacent `xsl:sort` elements which together define rules for sorting the items in an input sequence to form a sorted sequence.

sort key value

For each item in the initial sequence, a value is computed for each sort key component within the sort key specification. The value computed for an item by using the *N*th sort key component is referred to as the *N*th **sort key value** of that item.

sorted sequence

The sequence after sorting as defined by the `xsl:sort` elements is referred to as the **sorted sequence**.

source tree

The term **source tree** means any tree provided as input to the transformation. This includes the document containing the initial context node if any, documents containing nodes supplied as the values of stylesheet parameters, documents obtained from the results of functions such as `document`, `doc` [FO], and `collection` [FO], and documents returned by extension functions or extension instructions. In the context of a particular XSLT instruction, the term **source tree** means any tree provided as input to that instruction; this may be a source tree of the transformation as a whole, or it may be a temporary tree produced during the course of the transformation.

stable

A sort key specification is said to be **stable** if its first `xsl:sort` element has no `stable` attribute, or has a `stable` attribute whose effective value is `yes`.

standalone stylesheet module

A **standalone stylesheet module** is a stylesheet module that comprises the whole of an XML document.

standard attributes

There are a number of **standard attributes** that may appear on any XSLT element: specifically `version`, `exclude-result-prefixes`, `extension-element-prefixes`, `xpath-default-namespace`, `default-collation`, and `use-when`.

standard function namespace

The **standard function namespace** `http://www.w3.org/2005/xpath-functions` is used for functions in the function library defined in [Functions and Operators] and standard functions defined in this specification.

standard stylesheet module

A **standard stylesheet module** is a tree, or part of a tree, consisting of an `xsl:stylesheet` or `xsl:transform` element (see *3.6 Stylesheet Element*) together with its descendant nodes and associated attributes and namespaces.

static error

An error that is detected by examining a stylesheet before execution starts (that is, before the source document and values of stylesheet parameters are available) is referred to as a **static error**.

string value

The term **string value** is defined in Section 5.13 string-value Accessor [DM]. Every node has a string value. For example, the string value

of an element is the concatenation of the string values of all its descendant text nodes.

stylesheet

A transformation in the XSLT language is expressed in the form of a **stylesheet**, whose syntax is well-formed XML [XML 1.0] conforming to the Namespaces in XML Recommendation [Namespaces in XML 1.0].

stylesheet function

An `xsl:function` declaration declares the name, parameters, and implementation of a **stylesheet function** that can be called from any XPath expression within the stylesheet.

stylesheet level

A **stylesheet level** is a collection of stylesheet modules connected using `xsl:include` declarations: specifically, two stylesheet modules *A* and *B* are part of the same stylesheet level if one of them includes the other by means of an `xsl:include` declaration, or if there is a third stylesheet module *C* that is in the same stylesheet level as both *A* and *B*.

stylesheet module

A stylesheet consists of one or more **stylesheet modules**, each one forming all or part of an XML document.

stylesheet parameter

A top-level `xsl:param` element declares a **stylesheet parameter**. A stylesheet parameter is a global variable with the additional property that its value can be supplied by the caller when a transformation is initiated.

supplied value

The value of the variable is computed using the expression given in the `select` attribute or the contained sequence constructor, as described in *9.3 Values of Variables and Parameters*. This value is referred to as the **supplied value** of the variable.

target namespace URI

The namespace URI that is to be used in the result tree as a substitute for a literal namespace URI is called the **target namespace URI**.

template

An `xsl:template` declaration defines a **template**, which contains a sequence constructor for creating nodes and/or atomic values. A template can serve either as a template rule, invoked by matching nodes against a pattern, or as a named template, invoked explicitly by name. It is also possible for the same template to serve in both capacities.

template parameter

An `xsl:param` element may appear as a child of an `xsl:template` element, before any non-`xsl:param` children of that element. Such a parameter is known as a **template parameter**. A template parameter is a local variable with the additional property that its value can be set when the template is called, using any of the instructions `xsl:call-template`, `xsl:apply-templates`, `xsl:apply-imports`, or `xsl:next-match`.

template rule

A stylesheet contains a set of **template rules** (see *6 Template Rules*). A template rule has three parts: a pattern that is matched against nodes, a (possibly empty) set of template parameters, and a sequence constructor that is evaluated to produce a sequence of items.

temporary output state

The second of the two output states is called **temporary output** state. This state applies when instructions are writing to a temporary tree or any other non-final destination.

temporary tree

The term **temporary tree** means any tree that is neither a source tree nor a final result tree.

top-level

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

tunnel parameter

A parameter passed to a template may be defined as a **tunnel parameter**. Tunnel parameters have the property that they are automatically passed on by the called template to any further templates that it calls, and so on recursively.

type annotation

The term **type annotation** is used in this specification to refer to the value returned by the `dm:type-name` accessor of a node: see Section 5.14 type-name Accessor[DM].

type errors

Certain errors are classified as **type errors**. A type error occurs when the value supplied as input to an operation is of the wrong type for that operation, for example when an integer is supplied to an operation that expects a node.

typed value

The term **typed value** is defined in Section 5.15 typed-value Accessor[DM]. Every node except an element defined in the schema with element-only content has a typed value. For example, the typed value of an attribute of type `xs:IDREFS` is a sequence of zero or more `xs:IDREF` values.

user-defined data element

In addition to declarations, the `xsl:stylesheet` element may contain any element not from the XSLT namespace, provided that the expanded-QName of the element has a non-null namespace URI. Such elements are referred to as **user-defined data elements**.

value

A variable is a binding between a name and a value. The **value** of a variable is any sequence (of nodes and/or atomic values), as defined in [Data Model].

variable

The `xsl:variable` element declares a **variable**, which may be a global variable or a local variable.

variable-binding element

The two elements `xsl:variable` and `xsl:param` are referred to as **variable-binding elements**

whitespace text node

A **whitespace text node** is a text node whose content consists entirely of whitespace characters (that is, #x09, #x0A, #x0D, or #x20).

D Element Syntax Summary (Non-Normative)

The syntax of each XSLT element is summarized below, together with the context in the stylesheet where the element may appear. Some elements (specifically, instructions) are allowed as a child of any element that is allowed to contain a sequence constructor. These elements are:

- Literal result elements
- Extension instructions, if so defined

xsl:analyze-string

Category: instruction

Model:

```
<xsl:analyze-string
  select = expression
  regex = { string }
  flags? = { string }>
  <!-- Content: (xsl:matching-substring?, xsl:non-matching-substring?, xsl:fallback*) -->
</xsl:analyze-string>
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

xsl:apply-imports

Category: instruction

Model:

```
<xsl:apply-imports>
  <!-- Content: xsl:with-param* -->
</xsl:apply-imports>
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

xsl:apply-templates

Category: instruction

Model:

```
<xsl:apply-templates
  select? = expression
  mode? = token>
  <!-- Content: (xsl:sort | xsl:with-param)* -->
</xsl:apply-templates>
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*

- any literal result element

xsl:attribute

Category: instruction

Model:

```
<xsl:attribute
  name = { qname }
  namespace? = { uri-reference }
  select? = expression
  separator? = { string }
  type? = qname
  validation? = "strict" | "lax" | "preserve" | "strip">
  <!-- Content: sequence-constructor -->
</xsl:attribute>
```

Permitted parent elements:

- xsl:attribute-set
- any XSLT element whose content model is *sequence constructor*
- any literal result element

xsl:attribute-set

Category: declaration

Model:

```
<xsl:attribute-set
  name = qname
  use-attribute-sets? = qnames>
  <!-- Content: xsl:attribute* -->
</xsl:attribute-set>
```

Permitted parent elements:

- xsl:stylesheet
- xsl:transform

xsl:call-template

Category: instruction

Model:

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

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

xsl:character-map

Category: declaration

Model:

```
<xsl:character-map
  name = qname
  use-character-maps? = qnames>
  <!-- Content: (xsl:output-character*) -->
</xsl:character-map>
```

Permitted parent elements:

- xsl:stylesheet
- xsl:transform

xsl:choose

Category: instruction

Model:

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

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*

- any literal result element

xsl:comment

Category: instruction

Model:

```
<xsl:comment
  select? = expression>
  <!-- Content: sequence-constructor -->
</xsl:comment>
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

xsl:copy

Category: instruction

Model:

```
<xsl:copy
  copy-namespaces? = "yes" | "no"
  inherit-namespaces? = "yes" | "no"
  use-attribute-sets? = qnames
  type? = qname
  validation? = "strict" | "lax" | "preserve" | "strip">
  <!-- Content: sequence-constructor -->
</xsl:copy>
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

xsl:copy-of

Category: instruction

Model:

```
<xsl:copy-of
  select = expression
  copy-namespaces? = "yes" | "no"
  type? = qname
  validation? = "strict" | "lax" | "preserve" | "strip" />
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

xsl:decimal-format

Category: declaration

Model:

```
<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 />
```

Permitted parent elements:

- xsl:stylesheet
- xsl:transform

xsl:document

Category: instruction

Model:

```
<xsl:document
  validation? = "strict" | "lax" | "preserve" | "strip"
  type? = qname>
```

```
  <!-- Content: sequence-constructor -->
</xsl:document>
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

xsl:element

Category: instruction

Model:

```
<xsl:element
  name = { qname }
  namespace? = { uri-reference }
  inherit-namespaces? = "yes" | "no"
  use-attribute-sets? = qnames
  type? = qname
  validation? = "strict" | "lax" | "preserve" | "strip">
  <!-- Content: sequence-constructor -->
</xsl:element>
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

xsl:fallback

Category: instruction

Model:

```
<xsl:fallback>
  <!-- Content: sequence-constructor -->
</xsl:fallback>
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

xsl:for-each

Category: instruction

Model:

```
<xsl:for-each
  select = expression>
  <!-- Content: (xsl:sort*, sequence-constructor) -->
</xsl:for-each>
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

xsl:for-each-group

Category: instruction

Model:

```
<xsl:for-each-group
  select = expression
  group-by? = expression
  group-adjacent? = expression
  group-starting-with? = pattern
  group-ending-with? = pattern
  collation? = { uri }>
  <!-- Content: (xsl:sort*, sequence-constructor) -->
</xsl:for-each-group>
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

xsl:function

Category: declaration

Model:

```
<xsl:function
```

```
  name = qname
  as? = sequence-type
  override? = "yes" | "no">
  <!-- Content: (xsl:param*, sequence-constructor) -->
</xsl:function>
```

Permitted parent elements:

- xsl:stylesheet
- xsl:transform

xsl:if

Category: instruction

Model:

```
<xsl:if
  test = expression>
  <!-- Content: sequence-constructor -->
</xsl:if>
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

xsl:import

Category: declaration

Model:

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

Permitted parent elements:

- xsl:stylesheet
- xsl:transform

xsl:import-schema

Category: declaration

Model:

```
<xsl:import-schema
  namespace? = uri-reference
  schema-location? = uri-reference>
  <!-- Content: xs:schema? -->
</xsl:import-schema>
```

Permitted parent elements:

- xsl:stylesheet
- xsl:transform

xsl:include

Category: declaration

Model:

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

Permitted parent elements:

- xsl:stylesheet
- xsl:transform

xsl:key

Category: declaration

Model:

```
<xsl:key
  name = qname
  match = pattern
  use? = expression
  collation? = uri>
  <!-- Content: sequence-constructor -->
</xsl:key>
```

Permitted parent elements:

- xsl:stylesheet
- xsl:transform

xsl:matching-substring

Model:

```
<xsl:matching-substring>
  <!-- Content: sequence-constructor -->
</xsl:matching-substring>
```

Permitted parent elements:

- xsl:analyze-string

xsl:message

Category: instruction

Model:

```
<xsl:message
  select? = expression
  terminate? = { "yes" | "no" }>
  <!-- Content: sequence-constructor -->
</xsl:message>
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element
- xsl:function

xsl:namespace

Category: instruction

Model:

```
<xsl:namespace
  name = { ncname }
  select? = expression>
  <!-- Content: sequence-constructor -->
</xsl:namespace>
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

xsl:namespace-alias

Category: declaration

Model:

```
<xsl:namespace-alias
  stylesheet-prefix = prefix | "#default"
  result-prefix = prefix | "#default" />
```

Permitted parent elements:

- xsl:stylesheet
- xsl:transform

xsl:next-match

Category: instruction

Model:

```
<xsl:next-match>
  <!-- Content: (xsl:with-param | xsl:fallback)* -->
</xsl:next-match>
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

xsl:non-matching-substring

Model:

```
<xsl:non-matching-substring>
```

```
  <!-- Content: sequence-constructor -->
</xsl:non-matching-substring>
```

Permitted parent elements:

- xsl:analyze-string

xsl:number

Category: instruction

Model:

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

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

xsl:otherwise

Model:

```
<xsl:otherwise>
  <!-- Content: sequence-constructor -->
</xsl:otherwise>
```

Permitted parent elements:

- xsl:choose

xsl:output

Category: declaration

Model:

```
<xsl:output
  name? = qname
  method? = "xml" | "html" | "xhtml" | "text" | qname-but-not-ncname
  byte-order-mark? = "yes" | "no"
  cdata-section-elements? = qnames
  doctype-public? = string
  doctype-system? = string
  encoding? = string
  escape-uri-attributes? = "yes" | "no"
  include-content-type? = "yes" | "no"
  indent? = "yes" | "no"
  media-type? = string
  normalization-form? = "NFC" | "NFD" | "NFKC" | "NFKD" | "fully-normalized" | "none" | nmtoken
  omit-xml-declaration? = "yes" | "no"
  standalone? = "yes" | "no" | "omit"
  undeclare-prefixes? = "yes" | "no"
  use-character-maps? = qnames
  version? = nmtoken />
```

Permitted parent elements:

- xsl:stylesheet
- xsl:transform

xsl:output-character

Model:

```
<xsl:output-character
  character = char
  string = string />
```

Permitted parent elements:

- xsl:character-map

xsl:param

Category: declaration

Model:

```
<xsl:param
  name = qname
  select? = expression
  as? = sequence-type
  required? = "yes" | "no"
  tunnel? = "yes" | "no">
  <!-- Content: sequence-constructor -->
</xsl:param>
```

Permitted parent elements:

- xsl:stylesheet
- xsl:transform
- xsl:function
- xsl:template

xsl:perform-sort

Category: instruction

Model:

```
<xsl:perform-sort
  select? = expression>
  <!-- Content: (xsl:sort+, sequence-constructor) -->
</xsl:perform-sort>
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

xsl:preserve-space

Category: declaration

Model:

```
<xsl:preserve-space
  elements = tokens />
```

Permitted parent elements:

- xsl:stylesheet
- xsl:transform

xsl:processing-instruction

Category: instruction

Model:

```
<xsl:processing-instruction
  name = { ncname }
  select? = expression>
  <!-- Content: sequence-constructor -->
</xsl:processing-instruction>
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

xsl:result-document

Category: instruction

Model:

```
<xsl:result-document
  format? = { qname }
  href? = { uri-reference }
  validation? = "strict" | "lax" | "preserve" | "strip"
  type? = qname
  method? = { "xml" | "html" | "xhtml" | "text" | qname-but-not-ncname }
  byte-order-mark? = { "yes" | "no" }
  cdata-section-elements? = { qnames }
  doctype-public? = { string }
  doctype-system? = { string }
  encoding? = { string }
  escape-uri-attributes? = { "yes" | "no" }
  include-content-type? = { "yes" | "no" }
  indent? = { "yes" | "no" }
  media-type? = { string }
  normalization-form? = { "NFC" | "NFD" | "NFKC" | "NFKD" | "fully-normalized" | "none" | nmtoken }
  omit-xml-declaration? = { "yes" | "no" }
  standalone? = { "yes" | "no" | "omit" }
  undeclare-prefixes? = { "yes" | "no" }
```

```
  use-character-maps? = qnames
  output-version? = { nmtoken }>
  <!-- Content: sequence-constructor -->
</xsl:result-document>
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

xsl:sequence

Category: instruction

Model:

```
<xsl:sequence
  select = expression>
  <!-- Content: xsl:fallback* -->
</xsl:sequence>
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

xsl:sort

Model:

```
<xsl:sort
  select? = expression
  lang? = { nmtoken }
  order? = { "ascending" | "descending" }
  collation? = { uri }
  stable? = { "yes" | "no" }
  case-order? = { "upper-first" | "lower-first" }
  data-type? = { "text" | "number" | qname-but-not-ncname }>
  <!-- Content: sequence-constructor -->
</xsl:sort>
```

Permitted parent elements:

- xsl:apply-templates
- xsl:for-each
- xsl:for-each-group
- xsl:perform-sort

xsl:strip-space

Category: declaration

Model:

```
<xsl:strip-space
  elements = tokens />
```

Permitted parent elements:

- xsl:stylesheet
- xsl:transform

xsl:stylesheet

Model:

```
<xsl:stylesheet
  id? = id
  extension-element-prefixes? = tokens
  exclude-result-prefixes? = tokens
  version = number
  xpath-default-namespace? = uri
  default-validation? = "preserve" | "strip"
  default-collation? = uri-list
  input-type-annotations? = "preserve" | "strip" | "unspecified">
  <!-- Content: (xsl:import*, other-declarations) -->
</xsl:stylesheet>
```

Permitted parent elements:

- None

xsl:template

Category: declaration

Model:

```
<xsl:template
  match? = pattern
  name? = qname
  priority? = number
  mode? = tokens
  as? = sequence-type>
  <!-- Content: (xsl:param*, sequence-constructor) -->
</xsl:template>
```

Permitted parent elements:

- xsl:stylesheet
- xsl:transform

xsl:text

Category: instruction

Model:

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

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

xsl:transform

Model:

```
<xsl:transform
  id? = id
  extension-element-prefixes? = tokens
  exclude-result-prefixes? = tokens
  version = number
  xpath-default-namespace? = uri
  default-validation? = "preserve" | "strip"
  default-collation? = uri-list
  input-type-annotations? = "preserve" | "strip" | "unspecified">
  <!-- Content: (xsl:import*, other-declarations) -->
</xsl:transform>
```

Permitted parent elements:

- None

xsl:value-of

Category: instruction

Model:

```
<xsl:value-of
  select? = expression
  separator? = { string }
  [disable-output-escaping]? = "yes" | "no">
  <!-- Content: sequence-constructor -->
</xsl:value-of>
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

xsl:variable

Category: declaration instruction

Model:

```
<xsl:variable
  name = qname
  select? = expression
  as? = sequence-type>
  <!-- Content: sequence-constructor -->
</xsl:variable>
```

Permitted parent elements:

- xsl:stylesheet
- xsl:transform
- xsl:function
- any XSLT element whose content model is *sequence constructor*
- any literal result element

xsl:when

Model:

```
<xsl:when
  test = expression>
  <!-- Content: sequence-constructor -->
</xsl:when>
```

Permitted parent elements:

- xsl:choose

xsl:with-param

Model:

```
<xsl:with-param
  name = qname
  select? = expression
  as? = sequence-type
  tunnel? = "yes" | "no">
  <!-- Content: sequence-constructor -->
</xsl:with-param>
```

Permitted parent elements:

- xsl:apply-templates
- xsl:apply-imports
- xsl:call-template
- xsl:next-match

E Summary of Error Conditions (Non-Normative)

This appendix provides a summary of error conditions that a processor may signal. This list is not exhaustive or definitive. The errors are numbered for ease of reference, but there is no implication that an implementation MUST signal errors using these error codes, or that applications can test for these codes. Moreover, implementations are not REQUIRED to signal errors using the descriptive text used here.

Static errors

ERR XTSE0010

A static error is signaled if an XSLT-defined element is used in a context where it is not permitted, if a REQUIRED attribute is omitted, or if the content of the element does not correspond to the content that is allowed for the element.

ERR XTSE0020

It is a static error if an attribute (other than an attribute written using curly brackets in a position where an attribute value template is permitted) contains a value that is not one of the permitted values for that attribute.

ERR XTSE0080

It is a static error to use a reserved namespace in the name of a named template, a mode, an attribute set, a key, a decimal-format, a variable or parameter, a stylesheet function, a named output definition, or a character map.

ERR XTSE0090

It is a static error for an element from the XSLT namespace to have an attribute whose namespace is either null (that is, an attribute with an unprefixed name) or the XSLT namespace, other than attributes defined for the element in this document.

ERR XTSE0110

The value of the `version` attribute MUST be a number: specifically, it MUST be a a valid instance of the type `xs:decimal` as defined in [XML Schema Part 2].

ERR XTSE0120

An `xsl:stylesheet` element MUST NOT have any text node children.

ERR XTSE0125

It is a static error if the value of an `[xsl:]default-collation` attribute, after resolving against the base URI, contains no URI that the implementation recognizes as a collation URI.

ERR XTSE0130

It is a static error if the `xsl:stylesheet` element has a child element whose name has a null namespace URI.

ERR XTSE0150

A literal result element that is used as the outermost element of a simplified stylesheet module MUST have an `xsl:version` attribute.

ERR XTSE0165

It is a static error if the processor is not able to retrieve the resource identified by the URI reference [in the `href` attribute of

xsl:include or xsl:import] , or if the resource that is retrieved does not contain a stylesheet module conforming to this specification.

ERR XTSE0170

An xsl:include element MUST be a top-level element.

ERR XTSE0180

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

ERR XTSE0190

An xsl:import element MUST be a top-level element.

ERR XTSE0200

The xsl:import element children MUST precede all other element children of an xsl:stylesheet element, including any xsl:include element children and any user-defined data elements.

ERR XTSE0210

It is a static error if a stylesheet module directly or indirectly imports itself.

ERR XTSE0215

It is a static error if an xsl:import-schema element that contains an xs:schema element has a schema-location attribute, or if it has a namespace attribute that conflicts with the target namespace of the contained schema.

ERR XTSE0220

It is a static error if the synthetic schema document does not satisfy the constraints described in [XML Schema Part 1] (section 5.1, *Errors in Schema Construction and Structure*). This includes, without loss of generality, conflicts such as multiple definitions of the same name.

ERR XTSE0260

Within an XSLT element that is REQUIRED to be empty, any content other than comments or processing instructions, including any whitespace text node preserved using the xml:space="preserve" attribute, is a static error.

ERR XTSE0265

It is a static error if there is a stylesheet module in the stylesheet that specifies input-type-annotations="strip" and another stylesheet module that specifies input-type-annotations="preserve".

ERR XTSE0280

In the case of a prefixed QName used as the value of an attribute in the stylesheet, or appearing within an XPath expression in the stylesheet, it is a static error if the defining element has no namespace node whose name matches the prefix of the QName.

ERR XTSE0340

Where an attribute is defined to contain a pattern, it is a static error if the pattern does not match the production Pattern.

ERR XTSE0350

It is a static error if an unescaped left curly bracket appears in a fixed part of an attribute value template without a matching right curly bracket.

ERR XTSE0370

It is a static error if an unescaped right curly bracket occurs in a fixed part of an attribute value template.

ERR XTSE0500

An xsl:template element MUST have either a match attribute or a name attribute, or both. An xsl:template element that has no match attribute MUST have no mode attribute and no priority attribute.

ERR XTSE0530

The value of this attribute [the priority attribute of the xsl:template element] MUST conform to the rules for the xs:decimal type defined in [XML Schema Part 2]. Negative values are permitted..

ERR XTSE0550

It is a static error if the list [of modes in the mode attribute of xsl:template] is empty, if the same token is included more than once in the list, if the list contains an invalid token, or if the token #all appears together with any other value.

ERR XTSE0580

It is a static error if two parameters of a template or of a stylesheet function have the same name.

ERR XTSE0620

It is a static error if a variable-binding element has a select attribute and has non-empty content.

ERR XTSE0630

It is a static error if a stylesheet contains more than one binding of a global variable with the same name and same import precedence, unless it also contains another binding with the same name and higher import precedence.

ERR XTSE0650

It is a static error if a stylesheet contains an `xsl:call-template` instruction whose `name` attribute does not match the `name` attribute of any `xsl:template` in the stylesheet.

ERR XTSE0660

It is a static error if a stylesheet contains more than one template with the same name and the same import precedence, unless it also contains a template with the same name and higher import precedence.

ERR XTSE0670

It is a static error if a single `xsl:call-template`, `xsl:apply-templates`, `xsl:apply-imports`, or `xsl:next-match` element contains two or more `xsl:with-param` elements with matching `name` attributes.

ERR XTSE0680

In the case of `xsl:call-template`, it is a static error to pass a non-tunnel parameter named *x* to a template that does not have a template parameter named *x*, unless backwards compatible behavior is enabled for the `xsl:call-template` instruction.

ERR XTSE0690

It is a static error if a template that is invoked using `xsl:call-template` declares a template parameter specifying `required="yes"` and not specifying `tunnel="yes"`, if no value for this parameter is supplied by the calling instruction.

ERR XTSE0710

It is a static error if the value of the `use-attribute-sets` attribute of an `xsl:copy`, `xsl:element`, or `xsl:attribute-set` element, or the `xsl:use-attribute-sets` attribute of a literal result element, is not a whitespace-separated sequence of QNames, or if it contains a QName that does not match the `name` attribute of any `xsl:attribute-set` declaration in the stylesheet.

ERR XTSE0720

It is a static error if an `xsl:attribute-set` element directly or indirectly references itself via the names contained in the `use-attribute-sets` attribute.

ERR XTSE0740

A stylesheet function MUST have a prefixed name, to remove any risk of a clash with a function in the default function namespace. It is a static error if the name has no prefix.

ERR XTSE0760

Because arguments to a stylesheet function call MUST all be specified, the `xsl:param` elements within an `xsl:function` element MUST NOT specify a default value: this means they MUST be empty, and MUST NOT have a `select` attribute.

ERR XTSE0770

It is a static error for a stylesheet to contain two or more functions with the same expanded-QName, the same arity, and the same import precedence, unless there is another function with the same expanded-QName and arity, and a higher import precedence.

ERR XTSE0805

It is a static error if an attribute on a literal result element is in the XSLT namespace, unless it is one of the attributes explicitly defined in this specification.

ERR XTSE0808

It is a static error if a namespace prefix is used within the `[xsl:]exclude-result-prefixes` attribute and there is no namespace binding in scope for that prefix.

ERR XTSE0809

It is a static error if the value `#default` is used within the `[xsl:]exclude-result-prefixes` attribute and the parent element of the `[xsl:]exclude-result-prefixes` attribute has no default namespace.

ERR XTSE0810

It is a static error if there is more than one such declaration [more than one `xsl:namespace-alias` declaration] with the same literal namespace URI and the same import precedence and different values for the target namespace URI, unless there is also an `xsl:namespace-alias` declaration with the same literal namespace URI and a higher import precedence.

ERR XTSE0812

It is a static error if a value other than `#default` is specified for either the `stylesheet-prefix` or the `result-prefix` attributes of the `xsl:namespace-alias` element when there is no in-scope binding for that namespace prefix.

ERR XTSE0840

It is a static error if the `select` attribute of the `xsl:attribute` element is present unless the element has empty content.

ERR XTSE0870

It is a static error if the `select` attribute of the `xsl:value-of` element is present when the content of the element is non-empty, or if the `select` attribute is absent when the content is empty.

ERR XTSE0880

It is a static error if the `select` attribute of the `xsl:processing-instruction` element is present unless the element has empty content.

ERR XTSE0910

It is a static error if the `select` attribute of the `xsl:namespace` element is present when the element has content other than one or more `xsl:fallback` instructions, or if the `select` attribute is absent when the element has empty content.

ERR XTSE0940

It is a static error if the `select` attribute of the `xsl:comment` element is present unless the element has empty content.

ERR XTTE0950

It is a type error to use the `xsl:copy` or `xsl:copy-of` instruction to copy a node that has namespace-sensitive content if the `copy-namespaces` attribute has the value `no` and its explicit or implicit `validation` attribute has the value `preserve`. It is also a type error if either of these instructions (with `validation="preserve"`) is used to copy an attribute having namespace-sensitive content, unless the parent element is also copied. A node has namespace-sensitive content if its typed value contains an item of type `xs:QName` or `xs:NOTATION` or a type derived therefrom. The reason this is an error is because the validity of the content depends on the namespace context being preserved.

ERR XTSE0975

It is a static error if the `value` attribute of `xsl:number` is present unless the `select`, `level`, `count`, and `from` attributes are all absent.

ERR XTSE1015

It is a static error if an `xsl:sort` element with a `select` attribute has non-empty content.

ERR XTSE1017

It is a static error if an `xsl:sort` element other than the first in a sequence of sibling `xsl:sort` elements has a `stable` attribute.

ERR XTSE1040

It is a static error if an `xsl:perform-sort` instruction with a `select` attribute has any content other than `xsl:sort` and `xsl:fallback` instructions.

ERR XTSE1060

It is a static error if the `current-group` function is used within a pattern.

ERR XTSE1070

It is a static error if the `current-grouping-key` function is used within a pattern.

ERR XTSE1080

These four attributes [the `group-by`, `group-adjacent`, `group-starting-with`, and `group-ending-with` attributes of `xsl:for-each-group`] are mutually exclusive: it is a static error if none of these four attributes is present, or if more than one of them is present.

ERR XTSE1090

It is an error to specify the `collation` attribute if neither the `group-by` attribute nor `group-adjacent` attribute is specified.

ERR XTSE1130

It is a static error if the `xsl:analyze-string` instruction contains neither an `xsl:matching-substring` nor an `xsl:non-matching-substring` element.

ERR XTSE1205

It is a static error if an `xsl:key` declaration has a `use` attribute and has non-empty content, or if it has empty content and no `use` attribute.

ERR XTSE1210

It is a static error if the `xsl:key` declaration has a `collation` attribute whose value (after resolving against the base URI) is not a URI recognized by the implementation as referring to a collation.

ERR XTSE1220

It is a static error if there are several `xsl:key` declarations in the stylesheet with the same key name and different effective collations. Two collations are the same if their URIs are equal under the rules for comparing `xs:anyURI` values, or if the implementation can determine that they are different URIs referring to the same collation.

ERR XTSE1290

It is a static error if a named or unnamed decimal format contains two conflicting values for the same attribute in different `xsl:decimal-format` declarations having the same import precedence, unless there is another definition of the same attribute with higher import precedence.

ERR XTSE1295

It is a static error if the character specified in the `zero-digit` attribute is not a digit or is a digit that does not have the numeric value zero.

ERR XTSE1300

It is a static error if, for any named or unnamed decimal format, the variables representing characters used in a picture string do not each have distinct values. These variables are *decimal-separator-sign*, *grouping-sign*, *percent-sign*, *per-mille-sign*, *digit-zero-sign*, *digit-sign*, and *pattern-separator-sign*.

ERR XTSE1430

It is a static error if there is no namespace bound to the prefix on the element bearing the `[xsl:]extension-element-prefixes` attribute or, when `#default` is specified, if there is no default namespace.

ERR XTSE1505

It is a static error if both the `[xsl:]type` and `[xsl:]validation` attributes are present on the `xsl:element`, `xsl:attribute`, `xsl:copy`, `xsl:copy-of`, `xsl:document`, or `xsl:result-document` instructions, or on a literal result element.

ERR XTSE1520

It is a static error if the value of the `type` attribute of an `xsl:element`, `xsl:attribute`, `xsl:copy`, `xsl:copy-of`, `xsl:document`, or `xsl:result-document` instruction, or the `xsl:type` attribute of a literal result element, is not a valid `QName`, or if it uses a prefix that is not defined in an in-scope namespace declaration, or if the QName is not the name of a type definition included in the in-scope schema components for the stylesheet.

ERR XTSE1530

It is a static error if the value of the `type` attribute of an `xsl:attribute` instruction refers to a complex type definition

ERR XTSE1560

It is a static error if two `xsl:output` declarations within an output definition specify explicit values for the same attribute (other than `cdata-section-elements` and `use-character-maps`), with the values of the attributes being not equal, unless there is another `xsl:output` declaration within the same output definition that has higher import precedence and that specifies an explicit value for the same attribute.

ERR XTSE1570

The value [of the `method` attribute on `xsl:output`] MUST (if present) be a valid QName. If the QName does not have a prefix, then it identifies a method specified in [XSLT and XQuery Serialization] and MUST be one of `xml`, `html`, `xhtml`, or `text`.

ERR XTSE1580

It is a static error if the stylesheet contains two or more character maps with the same name and the same import precedence, unless it also contains another character map with the same name and higher import precedence.

ERR XTSE1590

It is a static error if a name in the `use-character-maps` attribute of the `xsl:output` or `xsl:character-map` elements does not match the `name` attribute of any `xsl:character-map` in the stylesheet.

ERR XTSE1600

It is a static error if a character map references itself, directly or indirectly, via a name in the `use-character-maps` attribute.

ERR XTSE1650

A basic XSLT processor MUST signal a static error if the stylesheet includes an `xsl:import-schema` declaration.

ERR XTSE1660

A basic XSLT processor MUST signal a static error if the stylesheet includes an `[xsl:]type` attribute, or an `[xsl:]validation` or `default-validation` attribute with a value other than `strip`.

Type errors

ERR XTTE0505

It is a type error if the result of evaluating the sequence constructor cannot be converted to the required type.

ERR XTTE0510

It is a type error if an `xsl:apply-templates` instruction with no `select` attribute is evaluated when the context item is not a node.

ERR XTTE0520

It is a type error if the sequence returned by the `select` expression [of `xsl:apply-templates`] contains an item that is not a node.

ERR XTTE0570

It is a type error if the supplied value of a variable cannot be converted to the required type.

ERR XTTE0590

It is a type error if the conversion of the supplied value of a parameter to its required type fails.

ERR XTTE0600

If a default value is given explicitly, that is, if there is either a `select` attribute or a non-empty sequence constructor, then it is a type error if the default value cannot be converted to the required type, using the function conversion rules.

ERR XTTE0780

If the `as` attribute [of `xsl:function`] is specified, then the result evaluated by the sequence constructor (see *5.7 Sequence Constructors*) is converted to the required type, using the function conversion rules. It is a type error if this conversion fails.

ERR XTTE0790

If the value of a parameter to a stylesheet function cannot be converted to the required type, a type error is signaled.

ERR XTTE0990

It is a type error if the `xsl:number` instruction is evaluated, with no `value` or `select` attribute, when the context item is not a node.

ERR XTTE1000

It is a type error if the result of evaluating the `select` attribute of the `xsl:number` instruction is anything other than a single node.

ERR XTTE1020

If any sort key value, after atomization and any type conversion REQUIRED by the `data-type` attribute, is a sequence containing more than one item, then the effect depends on whether the `xsl:sort` element is evaluated with backwards compatible behavior. With backwards compatible behavior, the effective sort key value is the first item in the sequence. In other cases, this is a type error.

ERR XTTE1100

It is a type error if the grouping key evaluated using the `group-adjacent` attribute is an empty sequence, or a sequence containing more than one item.

ERR XTTE1120

When the `group-starting-with` or `group-ending-with` attribute [of the `xsl:for-each-group` instruction] is used, it is a type error if the result of evaluating the `select` expression contains an item that is not a node.

ERR XTTE1510

If the `validation` attribute of an `xsl:element`, `xsl:attribute`, `xsl:copy`, `xsl:copy-of`, or `xsl:result-document` instruction, or the `xsl:validation` attribute of a literal result element, has the effective value `strict`, and schema validity assessment concludes that the validity of the element or attribute is invalid or unknown, a type error occurs. As with other type errors, the error MAY be signaled statically if it can be detected statically.

ERR XTTE1512

If the `validation` attribute of an `xsl:element`, `xsl:attribute`, `xsl:copy`, `xsl:copy-of`, or `xsl:result-document` instruction, or the `xsl:validation` attribute of a literal result element, has the effective value `strict`, and there is no matching top-level declaration in the schema, then a type error occurs. As with other type errors, the error MAY be signaled statically if it can be detected statically.

ERR XTTE1515

If the `validation` attribute of an `xsl:element`, `xsl:attribute`, `xsl:copy`, `xsl:copy-of`, or `xsl:result-document` instruction, or the `xsl:validation` attribute of a literal result element, has the effective value `lax`, and schema validity assessment concludes that the element or attribute is invalid, a type error occurs. As with other type errors, the error MAY be signaled statically if it can be detected statically.

ERR XTTE1540

It is a type error if an `[xsl:]type` attribute is defined for a constructed element or attribute, and the outcome of schema validity assessment against that type is that the `validity` property of that element or attribute information item is other than `valid`.

ERR XTTE1545

A type error occurs if a `type` or `validation` attribute is defined (explicitly or implicitly) for an instruction that constructs a new attribute node, if the effect of this is to cause the attribute value to be validated against a type that is derived from, or constructed by list or union from, the primitive types `xs:QName` or `xs:NOTATION`.

ERR XTTE1550

A type error occurs [when a document node is validated] unless the children of the document node comprise exactly one element node, no text nodes, and zero or more comment and processing instruction nodes, in any order.

ERR XTTE1555

It is a type error if, when validating a document node, document-level constraints are not satisfied. These constraints include identity constraints (`xs:unique`, `xs:key`, and `xs:keyref`) and ID/IDREF constraints.

Dynamic errors

ERR XTDE0030

It is a non-recoverable dynamic error if the effective value of an attribute written using curly brackets, in a position where an attribute value template is permitted, is a value that is not one of the permitted values for that attribute. If the processor is able to detect the error statically (for example, when any XPath expressions within the curly brackets can be evaluated statically), then the processor may optionally signal this as a static error.

ERR XTDE0040

It is a non-recoverable dynamic error if the invocation of the stylesheet specifies a template name that does not match the expanded-QName of a named template defined in the stylesheet.

ERR XTDE0045

It is a non-recoverable dynamic error if the invocation of the stylesheet specifies an initial mode (other than the default mode) that does not match the expanded-QName in the `mode` attribute of any template defined in the stylesheet.

ERR XTDE0047

It is a non-recoverable dynamic error if the invocation of the stylesheet specifies both an initial mode and an initial template.

ERR XTDE0050

It is a non-recoverable dynamic error if the stylesheet that is invoked declares a visible stylesheet parameter with `required="yes"` and no value for this parameter is supplied during the invocation of the stylesheet. A stylesheet parameter is visible if it is not masked by another global variable or parameter with the same name and higher import precedence.

ERR XTDE0060

It is a non-recoverable dynamic error if the initial template defines a template parameter that specifies `required="yes"`.

ERR XTDE0160

If an implementation does not support backwards-compatible behavior, then it is a non-recoverable dynamic error if any element is evaluated that enables backwards-compatible behavior.

ERR XTRE0270

It is a recoverable dynamic error if this [the process of finding an `xsl:strip-space` or `xsl:preserve-space` declaration to match an element in the source document] leaves more than one match, unless all the matched declarations are equivalent (that is, they are all `xsl:strip-space` or they are all `xsl:preserve-space`).

Action: The optional recovery action is to select, from the matches that are left, the one that occurs last in declaration order.

ERR XTDE0290

Where the result of evaluating an XPath expression (or an attribute value template) is required to be a lexical QName, then unless otherwise specified it is a non-recoverable dynamic error if the defining element has no namespace node whose name matches the prefix of the lexical QName. This error MAY be signaled as a static error if the value of the expression can be determined statically.

ERR XTDE0410

It is a non-recoverable dynamic error if the result sequence used to construct the content of an element node contains a namespace node or attribute node that is preceded in the sequence by a node that is neither a namespace node nor an attribute node.

ERR XTDE0420

It is a non-recoverable dynamic error if the result sequence used to construct the content of a document node contains a namespace node or attribute node.

ERR XTDE0430

It is a non-recoverable dynamic error if the result sequence contains two or more namespace nodes having the same name but different string values (that is, namespace nodes that map the same prefix to different namespace URIs).

ERR XTDE0440

It is a non-recoverable dynamic error if the result sequence contains a namespace node with no name and the element node being constructed has a null namespace URI (that is, it is an error to define a default namespace when the element is in no namespace).

ERR XTDE0485

It is a non-recoverable dynamic error if namespace fixup is performed on an element that contains among the typed values of the element and its attributes two values of type `xs:QName` or `xs:NOTATION` containing conflicting namespace prefixes, that is, two values that use the same prefix to refer to different namespace URIs.

ERR XTRE0540

It is a recoverable dynamic error if the conflict resolution algorithm for template rules leaves more than one matching template rule.

Action: The optional recovery action is to select, from the matching template rules that are left, the one that occurs last in declaration order.

ERR XTDE0560

It is a non-recoverable dynamic error if `xsl:apply-imports` or `xsl:next-match` is evaluated when the current template rule is null.

ERR XTDE0610

If an optional parameter has no `select` attribute and has an empty sequence constructor, and if there is an `as` attribute, then the default value of the parameter is an empty sequence. If the empty sequence is not a valid instance of the required type defined in the `as` attribute, then the parameter is treated as a required parameter, which means that it is a non-recoverable dynamic error if the caller supplies no value for the parameter.

ERR XTDE0640

In general, a circularity in a stylesheet is a non-recoverable dynamic error.

ERR XTDE0700

In other cases, [with `xsl:apply-templates`, `xsl:apply-imports`, and `xsl:next-match`, or `xsl:call-template` with tunnel parameters] it is a non-recoverable dynamic error if the template that is invoked declares a template parameter with `required="yes"` and no value for this parameter is supplied by the calling instruction.

ERR XTRE0795

It is a recoverable dynamic error if the name of a constructed attribute is `xml:space` and the value is not either `default` or `preserve`.
Action: The optional recovery action is to construct the attribute with the value as requested.

ERR XTDE0820

It is a non-recoverable dynamic error if the effective value of the `name` attribute [of the `xsl:element` instruction] is not a lexical QName.

ERR XTDE0830

In the case of an `xsl:element` instruction with no `namespace` attribute, it is a non-recoverable dynamic error if the effective value of the `name` attribute is a QName whose prefix is not declared in an in-scope namespace declaration for the `xsl:element` instruction.

ERR XTDE0835

It is a non-recoverable dynamic error if the effective value of the `namespace` attribute [of the `xsl:element` instruction] is not in the lexical space of the `xs:anyURI` data type.

ERR XTDE0850

It is a non-recoverable dynamic error if the effective value of the `name` attribute [of an `xsl:attribute` instruction] is not a lexical QName.

ERR XTDE0855

In the case of an `xsl:attribute` instruction with no `namespace` attribute, it is a non-recoverable dynamic error if the effective value of the `name` attribute is the string `xmlns`.

ERR XTDE0860

In the case of an `xsl:attribute` instruction with no `namespace` attribute, it is a non-recoverable dynamic error if the effective value of the `name` attribute is a lexical QName whose prefix is not declared in an in-scope namespace declaration for the `xsl:attribute` instruction.

ERR XTDE0865

It is a non-recoverable dynamic error if the effective value of the `namespace` attribute [of the `xsl:attribute` instruction] is not in the lexical space of the `xs:anyURI` data type.

ERR XTDE0890

It is a non-recoverable dynamic error if the effective value of the `name` attribute [of the `xsl:processing-instruction` instruction] is not both an NCName [Names] and a PITarget [XML].

ERR XTDE0905

It is a non-recoverable dynamic error if the string value of the new namespace node [created using `xsl:namespace`] is not valid in the lexical space of the data type `xs:anyURI`. [see ERR XTDE0835]

ERR XTDE0920

It is a non-recoverable dynamic error if the effective value of the `name` attribute [of the `xsl:namespace` instruction] is neither a zero-length string nor an NCName [Names], or if it is `xmlns`.

ERR XTDE0925

It is a non-recoverable dynamic error if the `xsl:namespace` instruction generates a namespace node whose name is `xml` and whose string value is not `http://www.w3.org/XML/1998/namespace`, or a namespace node whose string value is `http://www.w3.org/XML/1998/namespace` and whose name is not `xml`.

ERR XTDE0930

It is a non-recoverable dynamic error if evaluating the `select` attribute or the contained sequence constructor of an `xsl:namespace` instruction results in a zero-length string.

ERR XTDE0980

It is a non-recoverable dynamic error if any undiscarded item in the atomized sequence supplied as the value of the `value` attribute of `xsl:number` cannot be converted to an integer, or if the resulting integer is less than 0 (zero).

ERR XTDE1030

It is a non-recoverable dynamic error if, for any sort key component, the set of sort key values evaluated for all the items in the initial sequence, after any type conversion requested, contains a pair of ordinary values for which the result of the XPath `lt` operator is an error.

ERR XTDE1035

It is a non-recoverable dynamic error if the `collation` attribute of `xsl:sort` (after resolving against the base URI) is not a URI that is recognized by the implementation as referring to a collation.

ERR XTDE1110

It is a non-recoverable dynamic error if the collation URI specified to `xsl:for-each-group` (after resolving against the base URI) is a collation that is not recognized by the implementation. (For notes, [see ERR XTDE1035].)

ERR XTDE1140

It is a non-recoverable dynamic error if the effective value of the `regex` attribute [of the `xsl:analyze-string` instruction] does not conform to the REQUIRED syntax for regular expressions, as specified in [Functions and Operators]. If the regular expression is known statically (for example, if the attribute does not contain any expressions enclosed in curly brackets) then the processor MAY signal the error as a static error.

ERR XTDE1145

It is a non-recoverable dynamic error if the effective value of the `flags` attribute [of the `xsl:analyze-string` instruction] has a value other than the values defined in [Functions and Operators]. If the value of the attribute is known statically (for example, if the attribute does not contain any expressions enclosed in curly brackets) then the processor MAY signal the error as a static error.

ERR XTDE1150

It is a non-recoverable dynamic error if the effective value of the `regex` attribute [of the `xsl:analyze-string` instruction] is a regular expression that matches a zero-length string: or more specifically, if the regular expression `$r` and flags `$f` are such that `matches("", $r, $f)` returns true. If the regular expression is known statically (for example, if the attribute does not contain any expressions enclosed in curly brackets) then the processor MAY signal the error as a static error.

ERR XTRE1160

When a URI reference [supplied to the `document` function] contains a fragment identifier, it is a recoverable dynamic error if the media type is not one that is recognized by the processor, or if the fragment identifier does not conform to the rules for fragment identifiers for that media type, or if the fragment identifier selects something other than a sequence of nodes (for example, if it selects a range of characters within a text node).

Action: The optional recovery action is to ignore the fragment identifier and return the document node.

ERR XTDE1170

It is a non-recoverable dynamic error if a URI [supplied in the first argument to the `unparsed-text` function] contains a fragment identifier, or if it cannot be used to retrieve a resource containing text.

ERR XTDE1190

It is a non-recoverable dynamic error if a resource [retrieved using the `unparsed-text` function] contains octets that cannot be decoded into Unicode characters using the specified encoding, or if the resulting characters are not permitted XML characters. This includes the case where the processor does not support the requested encoding.

ERR XTDE1200

It is a non-recoverable dynamic error if the second argument of the `unparsed-text` function is omitted and the processor cannot infer the encoding using external information and the encoding is not UTF-8.

ERR XTDE1260

It is a non-recoverable dynamic error if the value [of the first argument to the `key` function] is not a valid QName, or if there is no namespace declaration in scope for the prefix of the QName, or if the name obtained by expanding the QName is not the same as the expanded name of any `xsl:key` declaration in the stylesheet. If the processor is able to detect the error statically (for example, when the argument is supplied as a string literal), then the processor MAY optionally signal this as a static error.

ERR XTDE1270

It is a non-recoverable dynamic error to call the `key` function with two arguments if there is no context node, or if the root of the tree containing the context node is not a document node; or to call the function with three arguments if the root of the tree containing the node supplied in the third argument is not a document node.

ERR XTDE1280

It is a non-recoverable dynamic error if the name specified as the `$decimal-format-name` argument [to the `format-number` function] is not a valid QName, or if its prefix has not been declared in an in-scope namespace declaration, or if the stylesheet does not contain a declaration of a decimal-format with a matching expanded-QName. If the processor is able to detect the error statically (for example, when the argument is supplied as a string literal), then the processor MAY optionally signal this as a static error.

ERR XTDE1310

The picture string [supplied to the `format-number` function] MUST conform to the following rules. [See full specification.] It is a non-recoverable dynamic error if the picture string does not satisfy these rules.

ERR XTDE1340

It is a non-recoverable dynamic error if the syntax of the picture [used for date/time formatting] is incorrect.

ERR XTDE1350

It is a non-recoverable dynamic error if a component specifier within the picture [used for date/time formatting] refers to components that are not available in the given type of `$value`, for example if the picture supplied to the `format-time` refers to the year, month, or day component.

ERR XTDE1360

If the `current` function is evaluated within an expression that is evaluated when the context item is undefined, a non-recoverable dynamic error occurs.

ERR XTDE1370

It is a non-recoverable dynamic error if the `unparsed-entity-uri` function is called when there is no context node, or when the root of the tree containing the context node is not a document node.

ERR XTDE1380

It is a non-recoverable dynamic error if the `unparsed-entity-public-id` function is called when there is no context node, or when the root of the tree containing the context node is not a document node.

ERR XTDE1390

It is a non-recoverable dynamic error if the value [supplied as the `$property-name` argument to the `system-property` function] is not a valid QName, or if there is no namespace declaration in scope for the prefix of the QName. If the processor is able to detect the error statically (for example, when the argument is supplied as a string literal), then the processor MAY optionally signal this as a static error.

ERR XTMM9000

When a transformation is terminated by use of `xsl:message terminate="yes"`, the effect is the same as when a non-recoverable dynamic error occurs during the transformation.

ERR XTDE1400

It is a non-recoverable dynamic error if the argument [passed to the `function-available` function] does not evaluate to a string that is a valid QName, or if there is no namespace declaration in scope for the prefix of the QName. If the processor is able to detect the error statically (for example, when the argument is supplied as a string literal), then the processor MAY optionally signal this as a static error.

ERR XTDE1420

It is a non-recoverable dynamic error if the arguments supplied to a call on an extension function do not satisfy the rules defined for that particular extension function, or if the extension function reports an error, or if the result of the extension function cannot be converted to an XPath value.

ERR XTDE1425

When backwards compatible behavior is enabled, it is a non-recoverable dynamic error to evaluate an extension function call if no implementation of the extension function is available.

ERR XTDE1428

It is a non-recoverable dynamic error if the argument [passed to the `type-available` function] does not evaluate to a string that is a valid QName, or if there is no namespace declaration in scope for the prefix of the QName. If the processor is able to detect the error statically (for example, when the argument is supplied as a string literal), then the processor MAY optionally signal this as a static error.

ERR XTDE1440

It is a non-recoverable dynamic error if the argument [passed to the `element-available` function] does not evaluate to a string that is a valid QName, or if there is no namespace declaration in scope for the prefix of the QName. If the processor is able to detect the error statically (for example, when the argument is supplied as a string literal), then the processor MAY optionally signal this as a static error.

ERR XTDE1450

When a processor performs fallback for an extension instruction that is not recognized, if the instruction element has one or more `xsl:fallback` children, then the content of each of the `xsl:fallback` children MUST be evaluated; it is a non-recoverable dynamic error if it has no `xsl:fallback` children.

ERR XTDE1460

It is a non-recoverable dynamic error if the effective value of the `format` attribute [of an `xsl:result-document` element] is not a valid lexical QName, or if it does not match the expanded-QName of an output definition in the stylesheet. If the processor is able to detect

the error statically (for example, when the `format` attribute contains no curly brackets), then the processor MAY optionally signal this as a static error.

ERR XTDE1480

It is a non-recoverable dynamic error to evaluate the `xsl:result-document` instruction in temporary output state.

ERR XTDE1490

It is a non-recoverable dynamic error for a transformation to generate two or more final result trees with the same URI.

ERR XTRE1495

It is a recoverable dynamic error for a transformation to generate two or more final result trees with URIs that identify the same physical resource. The optional recovery action is implementation-dependent, since it may be impossible for the processor to detect the error.

ERR XTRE1500

It is a recoverable dynamic error for a stylesheet to write to an external resource and read from the same resource during a single transformation, whether or not the same URI is used to access the resource in both cases.
Action: The optional recovery action is implementation-dependent: implementations are not REQUIRED to detect the error condition. Note that if the error is not detected, it is undefined whether the document that is read from the resource reflects its state before or after the result tree is written.

ERR XTRE1620

It is a recoverable dynamic error if an `xsl:value-of` or `xsl:text` instruction specifies that output escaping is to be disabled and the implementation does not support this.
Action: The optional recovery action is to ignore the `disable-output-escaping` attribute.

ERR XTRE1630

It is a recoverable dynamic error if an `xsl:value-of` or `xsl:text` instruction specifies that output escaping is to be disabled when writing to a final result tree that is not being serialized.
Action: The optional recovery action is to ignore the `disable-output-escaping` attribute.

ERR XTDE1665

A basic XSLT processor MUST raise a non-recoverable dynamic error if the input to the processor includes a node with a type annotation other than `xs:untyped` or `xs:untypedAtomic`, or an atomic value of a type other than those which a basic XSLT processor supports.

F Checklist of Implementation-Defined Features (Non-Normative)

This appendix provides a summary of XSLT language features whose effect is explicitly implementation-defined. The conformance rules (see *21 Conformance*) require vendors to provide documentation that explains how these choices have been exercised.

1. The way in which an XSLT processor is invoked, and the way in which values are supplied for the source document, starting node, stylesheet parameters, and base output URI, are implementation-defined. (See *2.3 Initiating a Transformation*)
2. The mechanisms for creating new extension instructions and extension functions are implementation-defined. (See *2.7 Extensibility*)
3. Where the specification provides a choice between signaling a dynamic error or recovering, the decision that is made (but not the recovery action itself) is implementation-defined. (See *2.9 Error Handling*)
4. It is implementation-defined whether type errors are signaled statically. (See *2.9 Error Handling*)
5. The set of namespaces that are specially recognized by the implementation (for example, for user-defined data elements, and extension attributes) is implementation-defined. (See *3.6.2 User-defined Data Elements*)
6. The effect of user-defined data elements whose name is in a namespace recognized by the implementation is implementation-defined. (See *3.6.2 User-defined Data Elements*)
7. It is implementation-defined whether an XSLT 2.0 processor supports backwards-compatible behavior. (See *3.8 Backwards-Compatible Processing*)
8. It is implementation-defined what forms of URI reference are acceptable in the `href` attribute of the `xsl:include` and `xsl:import` elements, for example, the URI schemes that may be used, the forms of fragment identifier that may be used, and the media types that are supported. (See *3.10.1 Locating Stylesheet Modules*)
9. An implementation may define mechanisms, above and beyond `xsl:import-schema` that allow schema components such as type definitions to be made available within a stylesheet. (See *3.13 Built-in Types*)
10. It is implementation-defined which versions of XML and XML Namespaces (1.0 and/or 1.1) are supported. (See *4.1 XML Versions*)
11. Limits on the value space of primitive data types, where not fixed by [XML Schema Part 2], are implementation-defined. (See *4.6 Limits*)
12. The implicit timezone for a transformation is implementation-defined. (See *5.4.3.2 Other components of the XPath Dynamic Context*)
13. If an `xml:id` attribute that has not been subjected to attribute value normalization is copied from a source tree to a result tree, it is implementation-defined whether attribute value normalization will be applied during the copy process. (See *11.9.1 Shallow Copy*)
14. The numbering sequences supported by the `xsl:number` instructions, beyond those defined in this specification, are implementation-defined. (See *12.3 Number to String Conversion Attributes*)
15. There MAY be implementation-defined upper bounds on the numbers that can be formatted by `xsl:number` using any particular numbering sequence. (See *12.3 Number to String Conversion Attributes*)
16. The set of languages for which numbering is supported by `xsl:number`, and the method of choosing a default language, are implementation-defined. (See *12.3 Number to String Conversion Attributes*)
17. If the `data-type` attribute of the `xsl:sort` element has a value other than `text` or `number`, the effect is implementation-defined. (See *13.1.2 Comparing Sort Key Values*)
18. The facilities for defining collations and allocating URIs to identify them are implementation-defined. (See *13.1.3 Sorting Using Collations*)

19. The algorithm used by `xsl:sort` to locate a collation, given the values of the `lang` and `case-order` attributes, is implementation-defined. (See *13.1.3 Sorting Using Collations*)
20. The set of media types recognized by the processor, for the purpose of interpreting fragment identifiers in URI references passed to the `document` function, is implementation-defined. (See *16.1 Multiple Source Documents*)
21. The set of encodings recognized by the `unparsed-text` function, other than `utf-8` and `utf-16`, is implementation-defined. (See *16.2 Reading Text Files*)
22. If no encoding is specified on a call to the `unparsed-text` function, the processor MAY use implementation-defined heuristics to determine the likely encoding. (See *16.2 Reading Text Files*)
23. The set of languages, calendars, and countries that are supported in the date formatting functions is implementation-defined. If any of these arguments is omitted or set to an empty sequence, the default is implementation-defined. (See *16.5.2 The Language, Calendar, and Country Arguments*)
24. The choice of the names and abbreviations used in any given language for calendar units such as days of the week and months of the year is implementation-defined. (See *16.5.2 The Language, Calendar, and Country Arguments*)
25. The values returned by the `system-property` function, and the names of the additional properties that are recognized, are implementation-defined. (See *16.6.5 system-property*)
26. The destination and formatting of messages written using the `xsl:message` instruction are implementation-defined. (See *17 Messages*)
27. The effect of an extension function returning a string containing characters that are not legal in XML is implementation-defined. (See *18.1.2 Calling Extension Functions*)
28. The way in which external objects are represented in the type system is implementation-defined. (See *18.1.3 External Objects*)
29. The way in which a final result tree is delivered to an application is implementation-defined. (See *19 Final Result Trees*)
30. Implementations MAY provide additional mechanisms allowing users to define the way in which final result trees are processed. (See *19.1 Creating Final Result Trees*)
31. If serialization is supported, then the location to which a final result tree is serialized is implementation-defined, subject to the constraint that relative URIs used to reference one tree from another remain valid. (See *20 Serialization*)
32. The default value of the `encoding` attribute of the `xsl:output` element is implementation-defined. (See *20 Serialization*)
33. It is implementation-defined which versions of XML, HTML, and XHTML are supported in the `version` attribute of the `xsl:output` declaration. (See *20 Serialization*)
34. The default value of the `byte-order-mark` serialization parameter is implementation-defined in the case of UTF-8 encoding. (See *20 Serialization*)
35. It is implementation-defined whether, and under what circumstances, disabling output escaping is supported. (See *20.2 Disabling Output Escaping*)

G Schema for XSLT Stylesheets (Non-Normative)

The following schema describes the structure of an XSLT stylesheet module. It does not define all the constraints that apply to a stylesheet (for example, it does not attempt to define a data type that precisely represents attributes containing XPath expressions). However, every valid stylesheet module conforms to this schema, unless it contains elements that invoke forwards-compatible-behavior.

A copy of this schema is available at http://www.w3.org/2007/schema-for-xslt20.xsd

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" targetNamespace="http://www.w3.org/1999/XSL/Transform" elementFormDe

<!-- ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ -->
<xs:annotation>
  <xs:documentation>

    This is a schema for XSLT 2.0 stylesheets.

    It defines all the elements that appear in the XSLT namespace; it also
    provides hooks that allow the inclusion of user-defined literal result elements,
    extension instructions, and top-level data elements.

    The schema is derived (with kind permission) from a schema for XSLT 1.0 stylesheets
    produced by Asir S Vedamuthu of WebMethods Inc.

    This schema is available for use under the conditions of the W3C Software License
    published at http://www.w3.org/Consortium/Legal/copyright-software-19980720

    The schema is organized as follows:

    PART A: definitions of complex types and model groups used as the basis
            for element definitions
    PART B: definitions of individual XSLT elements
    PART C: definitions for literal result elements
    PART D: definitions of simple types used in attribute definitions

    This schema does not attempt to define all the constraints that apply to a valid
    XSLT 2.0 stylesheet module. It is the intention that all valid stylesheet modules
    should conform to this schema; however, the schema is non-normative and in the event
    of any conflict, the text of the Recommendation takes precedence.

    This schema does not implement the special rules that apply when a stylesheet
    has sections that use forwards-compatible-mode. In this mode, setting version="3.0"
    allows elements from the XSLT namespace to be used that are not defined in XSLT 2.0.

    Simplified stylesheets (those with a literal result element as the outermost element)
    will validate against this schema only if validation starts in lax mode.

    This version is dated 2005-02-11
    Authors: Michael H Kay, Saxonica Limited
             Jeni Tennison, Jeni Tennison Consulting Ltd.

  </xs:documentation>
</xs:annotation>
<!-- ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ -->

<!--
The declaration of xml:space and xml:lang may need to be commented out because
```

```
of problems processing the schema using various tools
-->

<xs:import namespace="http://www.w3.org/XML/1998/namespace"
  schemaLocation="http://www.w3.org/2001/xml.xsd"/>

<!--
    An XSLT stylesheet may contain an in-line schema within an xsl:import-schema element,
    so the Schema for schemas needs to be imported
-->

<xs:import namespace="http://www.w3.org/2001/XMLSchema"
                  schemaLocation="http://www.w3.org/2001/XMLSchema.xsd"/>

<!-- ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ -->
<xs:annotation>
  <xs:documentation>
    PART A: definitions of complex types and model groups used as the basis
            for element definitions
  </xs:documentation>
</xs:annotation>
<!-- ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ -->

<xs:complexType name="generic-element-type" mixed="true">
  <xs:attribute name="default-collation" type="xsl:uri-list"/>
  <xs:attribute name="exclude-result-prefixes" type="xsl:prefix-list-or-all"/>
  <xs:attribute name="extension-element-prefixes" type="xsl:prefix-list"/>
  <xs:attribute name="use-when" type="xsl:expression"/>
  <xs:attribute name="xpath-default-namespace" type="xs:anyURI"/>
  <xs:anyAttribute namespace="##other" processContents="lax"/>
</xs:complexType>

<xs:complexType name="versioned-element-type" mixed="true">
  <xs:complexContent>
    <xs:extension base="xsl:generic-element-type">
      <xs:attribute name="version" type="xs:decimal" use="optional"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="element-only-versioned-element-type" mixed="false">
  <xs:complexContent>
    <xs:restriction base="xsl:versioned-element-type">
      <xs:anyAttribute namespace="##other" processContents="lax"/>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="sequence-constructor">
  <xs:complexContent mixed="true">
    <xs:extension base="xsl:versioned-element-type">
      <xs:group ref="xsl:sequence-constructor-group" minOccurs="0" maxOccurs="unbounded"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:group name="sequence-constructor-group">
  <xs:choice>
    <xs:element ref="xsl:variable"/>
    <xs:element ref="xsl:instruction"/>
    <xs:group ref="xsl:result-elements"/>
  </xs:choice>
</xs:group>

<xs:element name="declaration" type="xsl:generic-element-type" abstract="true"/>

<xs:element name="instruction" type="xsl:versioned-element-type" abstract="true"/>

<!-- ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ -->
<xs:annotation>
  <xs:documentation>
    PART B: definitions of individual XSLT elements
    Elements are listed in alphabetical order.
  </xs:documentation>
</xs:annotation>
<!-- ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ -->

<xs:element name="analyze-string" substitutionGroup="xsl:instruction">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="xsl:element-only-versioned-element-type">
        <xs:sequence>
          <xs:element ref="xsl:matching-substring" minOccurs="0"/>
          <xs:element ref="xsl:non-matching-substring" minOccurs="0"/>
          <xs:element ref="xsl:fallback" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="select" type="xsl:expression" use="required"/>
        <xs:attribute name="regex" type="xsl:avt" use="required"/>
        <xs:attribute name="flags" type="xsl:avt" default=""/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="apply-imports" substitutionGroup="xsl:instruction">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="xsl:element-only-versioned-element-type">
        <xs:sequence>
          <xs:element ref="xsl:with-param" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
```

```
  </xs:complexType>
</xs:element>

<xs:element name="apply-templates" substitutionGroup="xsl:instruction">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="xsl:element-only-versioned-element-type">
        <xs:choice minOccurs="0" maxOccurs="unbounded">
          <xs:element ref="xsl:sort"/>
          <xs:element ref="xsl:with-param"/>
        </xs:choice>
        <xs:attribute name="select" type="xsl:expression" default="child::node()"/>
        <xs:attribute name="mode" type="xsl:mode"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="attribute" substitutionGroup="xsl:instruction">
  <xs:complexType>
    <xs:complexContent mixed="true">
      <xs:extension base="xsl:sequence-constructor">
        <xs:attribute name="name" type="xsl:avt" use="required"/>
        <xs:attribute name="namespace" type="xsl:avt"/>
        <xs:attribute name="select" type="xsl:expression"/>
        <xs:attribute name="separator" type="xsl:avt"/>
        <xs:attribute name="type" type="xsl:QName"/>
        <xs:attribute name="validation" type="xsl:validation-type"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="attribute-set" substitutionGroup="xsl:declaration">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="xsl:element-only-versioned-element-type">
        <xs:sequence minOccurs="0" maxOccurs="unbounded">
          <xs:element ref="xsl:attribute"/>
        </xs:sequence>
        <xs:attribute name="name" type="xsl:QName" use="required"/>
        <xs:attribute name="use-attribute-sets" type="xsl:QNames" default=""/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="call-template" substitutionGroup="xsl:instruction">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="xsl:element-only-versioned-element-type">
        <xs:sequence>
          <xs:element ref="xsl:with-param" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="name" type="xsl:QName" use="required"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="character-map" substitutionGroup="xsl:declaration">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="xsl:element-only-versioned-element-type">
        <xs:sequence>
          <xs:element ref="xsl:output-character" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="name" type="xsl:QName" use="required"/>
        <xs:attribute name="use-character-maps" type="xsl:QNames" default=""/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="choose" substitutionGroup="xsl:instruction">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="xsl:element-only-versioned-element-type">
        <xs:sequence>
          <xs:element ref="xsl:when" maxOccurs="unbounded"/>
          <xs:element ref="xsl:otherwise" minOccurs="0"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="comment" substitutionGroup="xsl:instruction">
  <xs:complexType>
    <xs:complexContent mixed="true">
      <xs:extension base="xsl:sequence-constructor">
        <xs:attribute name="select" type="xsl:expression"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="copy" substitutionGroup="xsl:instruction">
  <xs:complexType>
    <xs:complexContent mixed="true">
      <xs:extension base="xsl:sequence-constructor">
        <xs:attribute name="copy-namespaces" type="xsl:yes-or-no" default="yes"/>
        <xs:attribute name="inherit-namespaces" type="xsl:yes-or-no" default="yes"/>
```

```
        <xs:attribute name="use-attribute-sets" type="xsl:QNames" default=""/>
        <xs:attribute name="type" type="xsl:QName"/>
        <xs:attribute name="validation" type="xsl:validation-type"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="copy-of" substitutionGroup="xsl:instruction">
  <xs:complexType>
    <xs:complexContent mixed="true">
      <xs:extension base="xsl:versioned-element-type">
        <xs:attribute name="select" type="xsl:expression" use="required"/>
        <xs:attribute name="copy-namespaces" type="xsl:yes-or-no" default="yes"/>
        <xs:attribute name="type" type="xsl:QName"/>
        <xs:attribute name="validation" type="xsl:validation-type"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="decimal-format" substitutionGroup="xsl:declaration">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="xsl:element-only-versioned-element-type">
        <xs:attribute name="name" type="xsl:QName"/>
        <xs:attribute name="decimal-separator" type="xsl:char" default="."/>
        <xs:attribute name="grouping-separator" type="xsl:char" default=","/>
        <xs:attribute name="infinity" type="xs:string" default="Infinity"/>
        <xs:attribute name="minus-sign" type="xsl:char" default="-"/>
        <xs:attribute name="NaN" type="xs:string" default="NaN"/>
        <xs:attribute name="percent" type="xsl:char" default="%"/>
        <xs:attribute name="per-mille" type="xsl:char" default="&#x2030;"/>
        <xs:attribute name="zero-digit" type="xsl:char" default="0"/>
        <xs:attribute name="digit" type="xsl:char" default="#"/>
        <xs:attribute name="pattern-separator" type="xsl:char" default=";"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="element" substitutionGroup="xsl:instruction">
  <xs:complexType mixed="true">
    <xs:complexContent>
      <xs:extension base="xsl:sequence-constructor">
        <xs:attribute name="name" type="xsl:avt" use="required"/>
        <xs:attribute name="namespace" type="xsl:avt"/>
        <xs:attribute name="inherit-namespaces" type="xsl:yes-or-no" default="yes"/>
        <xs:attribute name="use-attribute-sets" type="xsl:QNames" default=""/>
        <xs:attribute name="type" type="xsl:QName"/>
        <xs:attribute name="validation" type="xsl:validation-type"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="fallback" substitutionGroup="xsl:instruction" type="xsl:sequence-constructor"/>

<xs:element name="for-each" substitutionGroup="xsl:instruction">
  <xs:complexType>
    <xs:complexContent mixed="true">
      <xs:extension base="xsl:versioned-element-type">
        <xs:sequence>
          <xs:element ref="xsl:sort" minOccurs="0" maxOccurs="unbounded"/>
          <xs:group ref="xsl:sequence-constructor-group" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="select" type="xsl:expression" use="required"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="for-each-group" substitutionGroup="xsl:instruction">
  <xs:complexType>
    <xs:complexContent mixed="true">
      <xs:extension base="xsl:versioned-element-type">
        <xs:sequence>
          <xs:element ref="xsl:sort" minOccurs="0" maxOccurs="unbounded"/>
          <xs:group ref="xsl:sequence-constructor-group" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="select" type="xsl:expression" use="required"/>
        <xs:attribute name="group-by" type="xsl:expression"/>
        <xs:attribute name="group-adjacent" type="xsl:expression"/>
        <xs:attribute name="group-starting-with" type="xsl:pattern"/>
        <xs:attribute name="group-ending-with" type="xsl:pattern"/>
        <xs:attribute name="collation" type="xs:anyURI"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="function" substitutionGroup="xsl:declaration">
  <xs:complexType>
    <xs:complexContent mixed="true">
      <xs:extension base="xsl:versioned-element-type">
        <xs:sequence>
          <xs:element ref="xsl:param" minOccurs="0" maxOccurs="unbounded"/>
          <xs:group ref="xsl:sequence-constructor-group" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="name" type="xsl:QName" use="required"/>
        <xs:attribute name="override" type="xsl:yes-or-no" default="yes"/>
        <xs:attribute name="as" type="xsl:sequence-type" default="item()*"/>
      </xs:extension>
```

```
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="if" substitutionGroup="xsl:instruction">
  <xs:complexType>
    <xs:complexContent mixed="true">
      <xs:extension base="xsl:sequence-constructor">
        <xs:attribute name="test" type="xsl:expression" use="required"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="import">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="xsl:element-only-versioned-element-type">
        <xs:attribute name="href" type="xs:anyURI" use="required"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="import-schema" substitutionGroup="xsl:declaration">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="xsl:element-only-versioned-element-type">
        <xs:sequence>
          <xs:element ref="xs:schema" minOccurs="0" maxOccurs="1"/>
        </xs:sequence>
        <xs:attribute name="namespace" type="xs:anyURI"/>
        <xs:attribute name="schema-location" type="xs:anyURI"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="include" substitutionGroup="xsl:declaration">
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="xsl:element-only-versioned-element-type">
          <xs:attribute name="href" type="xs:anyURI" use="required"/>
        </xs:extension>
      </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="key" substitutionGroup="xsl:declaration">
  <xs:complexType>
    <xs:complexContent mixed="true">
      <xs:extension base="xsl:sequence-constructor">
        <xs:attribute name="name" type="xsl:QName" use="required"/>
        <xs:attribute name="match" type="xsl:pattern" use="required"/>
        <xs:attribute name="use" type="xsl:expression"/>
        <xs:attribute name="collation" type="xs:anyURI"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="matching-substring" type="xsl:sequence-constructor"/>

<xs:element name="message" substitutionGroup="xsl:instruction">
  <xs:complexType>
    <xs:complexContent mixed="true">
      <xs:extension base="xsl:sequence-constructor">
        <xs:attribute name="select" type="xsl:expression"/>
        <xs:attribute name="terminate" type="xsl:avt" default="no"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="namespace" substitutionGroup="xsl:instruction">
  <xs:complexType>
    <xs:complexContent mixed="true">
      <xs:extension base="xsl:sequence-constructor">
        <xs:attribute name="name" type="xsl:avt" use="required"/>
        <xs:attribute name="select" type="xsl:expression"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="namespace-alias" substitutionGroup="xsl:declaration">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="xsl:element-only-versioned-element-type">
        <xs:attribute name="stylesheet-prefix" type="xsl:prefix-or-default" use="required"/>
        <xs:attribute name="result-prefix" type="xsl:prefix-or-default" use="required"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="next-match" substitutionGroup="xsl:instruction">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="xsl:element-only-versioned-element-type">
        <xs:choice minOccurs="0" maxOccurs="unbounded">
          <xs:element ref="xsl:with-param"/>
          <xs:element ref="xsl:fallback"/>
```

```
        </xs:choice>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="non-matching-substring" type="xsl:sequence-constructor"/>

<xs:element name="number" substitutionGroup="xsl:instruction">
  <xs:complexType>
    <xs:complexContent mixed="true">
      <xs:extension base="xsl:versioned-element-type">
        <xs:attribute name="value" type="xsl:expression"/>
        <xs:attribute name="select" type="xsl:expression"/>
        <xs:attribute name="level" type="xsl:level" default="single"/>
        <xs:attribute name="count" type="xsl:pattern"/>
        <xs:attribute name="from" type="xsl:pattern"/>
        <xs:attribute name="format" type="xsl:avt" default="1"/>
        <xs:attribute name="lang" type="xsl:avt"/>
        <xs:attribute name="letter-value" type="xsl:avt"/>
        <xs:attribute name="ordinal" type="xsl:avt"/>
        <xs:attribute name="grouping-separator" type="xsl:avt"/>
        <xs:attribute name="grouping-size" type="xsl:avt"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="otherwise" type="xsl:sequence-constructor"/>

<xs:element name="output" substitutionGroup="xsl:declaration">
  <xs:complexType>
    <xs:complexContent mixed="true">
      <xs:extension base="xsl:generic-element-type">
        <xs:attribute name="name" type="xsl:QName"/>
        <xs:attribute name="method" type="xsl:method"/>
        <xs:attribute name="byte-order-mark" type="xsl:yes-or-no"/>
        <xs:attribute name="cdata-section-elements" type="xsl:QNames"/>
        <xs:attribute name="doctype-public" type="xs:string"/>
        <xs:attribute name="doctype-system" type="xs:string"/>
        <xs:attribute name="encoding" type="xs:string"/>
        <xs:attribute name="escape-uri-attributes" type="xsl:yes-or-no"/>
        <xs:attribute name="include-content-type" type="xsl:yes-or-no"/>
        <xs:attribute name="indent" type="xsl:yes-or-no"/>
        <xs:attribute name="media-type" type="xs:string"/>
        <xs:attribute name="normalization-form" type="xs:NMTOKEN"/>
        <xs:attribute name="omit-xml-declaration" type="xsl:yes-or-no"/>
        <xs:attribute name="standalone" type="xsl:yes-or-no-or-omit"/>
        <xs:attribute name="undeclare-prefixes" type="xsl:yes-or-no"/>
        <xs:attribute name="use-character-maps" type="xsl:QNames"/>
        <xs:attribute name="version" type="xs:NMTOKEN"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="output-character">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="xsl:element-only-versioned-element-type">
        <xs:attribute name="character" type="xsl:char" use="required"/>
        <xs:attribute name="string" type="xs:string" use="required"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="param">
  <xs:complexType>
    <xs:complexContent mixed="true">
      <xs:extension base="xsl:sequence-constructor">
        <xs:attribute name="name" type="xsl:QName" use="required"/>
        <xs:attribute name="select" type="xsl:expression"/>
        <xs:attribute name="as" type="xsl:sequence-type"/>
        <xs:attribute name="required" type="xsl:yes-or-no"/>
        <xs:attribute name="tunnel" type="xsl:yes-or-no"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="perform-sort" substitutionGroup="xsl:instruction">
  <xs:complexType>
    <xs:complexContent mixed="true">
      <xs:extension base="xsl:versioned-element-type">
        <xs:sequence>
          <xs:element ref="xsl:sort" minOccurs="1" maxOccurs="unbounded"/>
          <xs:group ref="xsl:sequence-constructor-group" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="select" type="xsl:expression"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="preserve-space" substitutionGroup="xsl:declaration">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="xsl:element-only-versioned-element-type">
        <xs:attribute name="elements" type="xsl:nametests" use="required"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
```

```
</xs:element>

<xs:element name="processing-instruction" substitutionGroup="xsl:instruction">
  <xs:complexType>
    <xs:complexContent mixed="true">
      <xs:extension base="xsl:sequence-constructor">
        <xs:attribute name="name" type="xsl:avt" use="required"/>
        <xs:attribute name="select" type="xsl:expression"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="result-document" substitutionGroup="xsl:instruction">
  <xs:complexType>
    <xs:complexContent mixed="true">
      <xs:extension base="xsl:sequence-constructor">
        <xs:attribute name="format" type="xsl:avt"/>
        <xs:attribute name="href" type="xsl:avt"/>
        <xs:attribute name="type" type="xsl:QName"/>
        <xs:attribute name="validation" type="xsl:validation-type"/>
        <xs:attribute name="method" type="xsl:avt"/>
        <xs:attribute name="byte-order-mark" type="xsl:avt"/>
        <xs:attribute name="cdata-section-elements" type="xsl:avt"/>
        <xs:attribute name="doctype-public" type="xsl:avt"/>
        <xs:attribute name="doctype-system" type="xsl:avt"/>
        <xs:attribute name="encoding" type="xsl:avt"/>
        <xs:attribute name="escape-uri-attributes" type="xsl:avt"/>
        <xs:attribute name="include-content-type" type="xsl:avt"/>
        <xs:attribute name="indent" type="xsl:avt"/>
        <xs:attribute name="media-type" type="xsl:avt"/>
        <xs:attribute name="normalization-form" type="xsl:avt"/>
        <xs:attribute name="omit-xml-declaration" type="xsl:avt"/>
        <xs:attribute name="standalone" type="xsl:avt"/>
        <xs:attribute name="undeclare-prefixes" type="xsl:avt"/>
        <xs:attribute name="use-character-maps" type="xsl:QNames"/>
        <xs:attribute name="output-version" type="xsl:avt"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="sequence" substitutionGroup="xsl:instruction">
  <xs:complexType>
    <xs:complexContent mixed="true">
      <xs:extension base="xsl:sequence-constructor">
        <xs:attribute name="select" type="xsl:expression"/>
        <xs:attribute name="as" type="xsl:sequence-type"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="sort">
  <xs:complexType>
    <xs:complexContent mixed="true">
      <xs:extension base="xsl:sequence-constructor">
        <xs:attribute name="select" type="xsl:expression"/>
        <xs:attribute name="lang" type="xsl:avt"/>
        <xs:attribute name="data-type" type="xsl:avt" default="text"/>
        <xs:attribute name="order" type="xsl:avt" default="ascending"/>
        <xs:attribute name="case-order" type="xsl:avt"/>
        <xs:attribute name="collation" type="xsl:avt"/>
        <xs:attribute name="stable" type="xsl:yes-or-no"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="strip-space" substitutionGroup="xsl:declaration">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="xsl:element-only-versioned-element-type">
        <xs:attribute name="elements" type="xsl:nametests" use="required"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="stylesheet" substitutionGroup="xsl:transform"/>

<xs:element name="template" substitutionGroup="xsl:declaration">
  <xs:complexType>
    <xs:complexContent mixed="true">
      <xs:extension base="xsl:versioned-element-type">
        <xs:sequence>
          <xs:element ref="xsl:param" minOccurs="0" maxOccurs="unbounded"/>
          <xs:group ref="xsl:sequence-constructor-group" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="match" type="xsl:pattern"/>
        <xs:attribute name="priority" type="xs:decimal"/>
        <xs:attribute name="mode" type="xsl:modes"/>
        <xs:attribute name="name" type="xsl:QName"/>
        <xs:attribute name="as" type="xsl:sequence-type" default="item()*"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:complexType name="text-element-base-type">
  <xs:simpleContent>
    <xs:restriction base="xsl:versioned-element-type">
      <xs:simpleType>
```

```
        <xs:restriction base="xs:string"/>
      </xs:simpleType>
      <xs:anyAttribute namespace="##other" processContents="lax"/>
    </xs:restriction>
  </xs:simpleContent>
</xs:complexType>

<xs:element name="text" substitutionGroup="xsl:instruction">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xsl:text-element-base-type">
        <xs:attribute name="disable-output-escaping" type="xsl:yes-or-no" default="no"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>

<xs:complexType name="transform-element-base-type">
  <xs:complexContent>
    <xs:restriction base="xsl:element-only-versioned-element-type">
      <xs:attribute name="version" type="xs:decimal" use="required"/>
      <xs:anyAttribute namespace="##other" processContents="lax"/>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>

<xs:element name="transform">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="xsl:transform-element-base-type">
        <xs:sequence>
          <xs:element ref="xsl:import" minOccurs="0" maxOccurs="unbounded"/>
          <xs:choice minOccurs="0" maxOccurs="unbounded">
            <xs:element ref="xsl:declaration"/>
            <xs:element ref="xsl:variable"/>
            <xs:element ref="xsl:param"/>
            <xs:any namespace="##other" processContents="lax"/> <!-- weaker than XSLT 1.0 -->
          </xs:choice>
        </xs:sequence>
        <xs:attribute name="id" type="xs:ID"/>
        <xs:attribute name="default-validation" type="xsl:validation-strip-or-preserve" default="strip"/>
        <xs:attribute name="input-type-annotations" type="xsl:input-type-annotations-type" default="unspecified"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="value-of" substitutionGroup="xsl:instruction">
  <xs:complexType>
    <xs:complexContent mixed="true">
      <xs:extension base="xsl:sequence-constructor">
        <xs:attribute name="select" type="xsl:expression"/>
        <xs:attribute name="separator" type="xsl:avt"/>
        <xs:attribute name="disable-output-escaping" type="xsl:yes-or-no" default="no"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="variable">
  <xs:complexType>
    <xs:complexContent mixed="true">
      <xs:extension base="xsl:sequence-constructor">
        <xs:attribute name="name" type="xsl:QName" use="required"/>
        <xs:attribute name="select" type="xsl:expression" use="optional"/>
        <xs:attribute name="as" type="xsl:sequence-type" use="optional"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="when">
  <xs:complexType>
    <xs:complexContent mixed="true">
      <xs:extension base="xsl:sequence-constructor">
        <xs:attribute name="test" type="xsl:expression" use="required"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="with-param">
  <xs:complexType>
    <xs:complexContent mixed="true">
      <xs:extension base="xsl:sequence-constructor">
        <xs:attribute name="name" type="xsl:QName" use="required"/>
        <xs:attribute name="select" type="xsl:expression"/>
        <xs:attribute name="as" type="xsl:sequence-type"/>
        <xs:attribute name="tunnel" type="xsl:yes-or-no"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<!-- ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ -->
<xs:annotation>
  <xs:documentation>
    PART C: definition of literal result elements

    There are three ways to define the literal result elements
    permissible in a stylesheet.

    (a) do nothing. This allows any element to be used as a literal
```

```
        result element, provided it is not in the XSLT namespace

    (b) declare all permitted literal result elements as members
        of the xsl:literal-result-element substitution group

    (c) redefine the model group xsl:result-elements to accommodate
        all permitted literal result elements.

    Literal result elements are allowed to take certain attributes
    in the XSLT namespace. These are defined in the attribute group
    literal-result-element-attributes, which can be included in the
    definition of any literal result element.

  </xs:documentation>
</xs:annotation>
<!-- ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ -->

<xs:element name="literal-result-element" abstract="true" type="xs:anyType"/>

<xs:attributeGroup name="literal-result-element-attributes">
  <xs:attribute name="default-collation" form="qualified" type="xsl:uri-list"/>
  <xs:attribute name="extension-element-prefixes" form="qualified" type="xsl:prefixes"/>
  <xs:attribute name="exclude-result-prefixes" form="qualified" type="xsl:prefixes"/>
  <xs:attribute name="xpath-default-namespace" form="qualified" type="xs:anyURI"/>
  <xs:attribute name="inherit-namespaces" form="qualified" type="xsl:yes-or-no" default="yes"/>
  <xs:attribute name="use-attribute-sets" form="qualified" type="xsl:QNames" default=""/>
  <xs:attribute name="use-when" form="qualified" type="xsl:expression"/>
  <xs:attribute name="version" form="qualified" type="xs:decimal"/>
  <xs:attribute name="type" form="qualified" type="xsl:QName"/>
  <xs:attribute name="validation" form="qualified" type="xsl:validation-type"/>
</xs:attributeGroup>

<xs:group name="result-elements">
  <xs:choice>
    <xs:element ref="xsl:literal-result-element"/>
    <xs:any namespace="##other" processContents="lax"/>
    <xs:any namespace="##local" processContents="lax"/>
  </xs:choice>
</xs:group>


<!-- ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ -->
<xs:annotation>
  <xs:documentation>
    PART D: definitions of simple types used in stylesheet attributes
  </xs:documentation>
</xs:annotation>
<!-- ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ -->

<xs:simpleType name="avt">
  <xs:annotation>
    <xs:documentation>
      This type is used for all attributes that allow an attribute value template.
      The general rules for the syntax of attribute value templates, and the specific
      rules for each such attribute, are described in the XSLT 2.0 Recommendation.
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:string"/>
</xs:simpleType>

<xs:simpleType name="char">
  <xs:annotation>
    <xs:documentation>
      A string containing exactly one character.
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:string">
    <xs:length value="1"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="expression">
  <xs:annotation>
    <xs:documentation>
      An XPath 2.0 expression.
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:token">
    <xs:pattern value=".+"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="input-type-annotations-type">
  <xs:annotation>
    <xs:documentation>
      Describes how type annotations in source documents are handled.
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:token">
    <xs:enumeration value="preserve"/>
    <xs:enumeration value="strip"/>
    <xs:enumeration value="unspecified"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="level">
  <xs:annotation>
    <xs:documentation>
      The level attribute of xsl:number:
      one of single, multiple, or any.
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:NCName">
```

```
    <xs:enumeration value="single"/>
    <xs:enumeration value="multiple"/>
    <xs:enumeration value="any"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="mode">
  <xs:annotation>
    <xs:documentation>
      The mode attribute of xsl:apply-templates:
      either a QName, or #current, or #default.
    </xs:documentation>
  </xs:annotation>
  <xs:union memberTypes="xsl:QName">
    <xs:simpleType>
      <xs:restriction base="xs:token">
        <xs:enumeration value="#default"/>
        <xs:enumeration value="#current"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:union>
</xs:simpleType>

<xs:simpleType name="modes">
  <xs:annotation>
    <xs:documentation>
      The mode attribute of xsl:template:
      either a list, each member being either a QName or #default;
      or the value #all
    </xs:documentation>
  </xs:annotation>
  <xs:union>
    <xs:simpleType>
      <xs:list>
        <xs:simpleType>
          <xs:union memberTypes="xsl:QName">
            <xs:simpleType>
              <xs:restriction base="xs:token">
                <xs:enumeration value="#default"/>
              </xs:restriction>
            </xs:simpleType>
          </xs:union>
        </xs:simpleType>
      </xs:list>
    </xs:simpleType>
    <xs:simpleType>
      <xs:restriction base="xs:token">
        <xs:enumeration value="#all"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:union>
</xs:simpleType>

<xs:simpleType name="nametests">
  <xs:annotation>
    <xs:documentation>
      A list of NameTests, as defined in the XPath 2.0 Recommendation.
      Each NameTest is either a QName, or "*", or "prefix:*", or "*:localname"
    </xs:documentation>
  </xs:annotation>
  <xs:list>
    <xs:simpleType>
      <xs:union memberTypes="xsl:QName">
        <xs:simpleType>
          <xs:restriction base="xs:token">
            <xs:enumeration value="*"/>
          </xs:restriction>
        </xs:simpleType>
        <xs:simpleType>
          <xs:restriction base="xs:token">
            <xs:pattern value="\i\c*:\*"/>
            <xs:pattern value="\*:\i\c*"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:union>
    </xs:simpleType>
  </xs:list>
</xs:simpleType>

<xs:simpleType name="prefixes">
  <xs:list itemType="xs:NCName"/>
</xs:simpleType>

<xs:simpleType name="prefix-list-or-all">
  <xs:union memberTypes="xsl:prefix-list">
    <xs:simpleType>
      <xs:restriction base="xs:token">
        <xs:enumeration value="#all"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:union>
</xs:simpleType>

<xs:simpleType name="prefix-list">
  <xs:list itemType="xsl:prefix-or-default"/>
</xs:simpleType>

<xs:simpleType name="method">
  <xs:annotation>
    <xs:documentation>
      The method attribute of xsl:output:
      Either one of the recognized names "xml", "xhtml", "html", "text",
      or a QName that must include a prefix.
```

```
    </xs:documentation>
  </xs:annotation>
  <xs:union>
    <xs:simpleType>
      <xs:restriction base="xs:NCName">
        <xs:enumeration value="xml"/>
        <xs:enumeration value="xhtml"/>
        <xs:enumeration value="html"/>
        <xs:enumeration value="text"/>
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType>
      <xs:restriction base="xsl:QName">
        <xs:pattern value="\c*:\c*"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:union>
</xs:simpleType>

<xs:simpleType name="pattern">
  <xs:annotation>
    <xs:documentation>
      A match pattern as defined in the XSLT 2.0 Recommendation.
      The syntax for patterns is a restricted form of the syntax for
      XPath 2.0 expressions.
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xsl:expression"/>
</xs:simpleType>

<xs:simpleType name="prefix-or-default">
  <xs:annotation>
    <xs:documentation>
      Either a namespace prefix, or #default.
      Used in the xsl:namespace-alias element.
    </xs:documentation>
  </xs:annotation>
  <xs:union memberTypes="xs:NCName">
    <xs:simpleType>
      <xs:restriction base="xs:token">
        <xs:enumeration value="#default"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:union>
</xs:simpleType>

<xs:simpleType name="QNames">
  <xs:annotation>
    <xs:documentation>
      A list of QNames.
      Used in the [xsl:]use-attribute-sets attribute of various elements,
      and in the cdata-section-elements attribute of xsl:output
    </xs:documentation>
  </xs:annotation>
  <xs:list itemType="xsl:QName"/>
</xs:simpleType>

<xs:simpleType name="QName">
  <xs:annotation>
    <xs:documentation>
      A QName.
      This schema does not use the built-in type xs:QName, but rather defines its own
      QName type. Although xs:QName would define the correct validation on these attributes,
      a schema processor would expand unprefixed QNames incorrectly when constructing the PSVI,
      because (as defined in XML Schema errata) an unprefixed xs:QName is assumed to be in
      the default namespace, which is not the correct assumption for XSLT.
      The data type is defined as a restriction of the built-in type Name, restricted
      so that it can only contain one colon which must not be the first or last character.
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:Name">
    <xs:pattern value="([^:]+:)?[^:]+"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="sequence-type">
  <xs:annotation>
    <xs:documentation>
      The description of a data type, conforming to the
      SequenceType production defined in the XPath 2.0 Recommendation
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:token">
    <xs:pattern value=".+"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="uri-list">
  <xs:list itemType="xs:anyURI"/>
</xs:simpleType>

<xs:simpleType name="validation-strip-or-preserve">
  <xs:annotation>
    <xs:documentation>
      Describes different ways of type-annotating an element or attribute.
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xsl:validation-type">
    <xs:enumeration value="preserve"/>
    <xs:enumeration value="strip"/>
  </xs:restriction>
</xs:simpleType>
```

```
<xs:simpleType name="validation-type">
  <xs:annotation>
    <xs:documentation>
      Describes different ways of type-annotating an element or attribute.
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:token">
    <xs:enumeration value="strict"/>
    <xs:enumeration value="lax"/>
    <xs:enumeration value="preserve"/>
    <xs:enumeration value="strip"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="yes-or-no">
  <xs:annotation>
    <xs:documentation>
      One of the values "yes" or "no".
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:token">
    <xs:enumeration value="yes"/>
    <xs:enumeration value="no"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="yes-or-no-or-omit">
  <xs:annotation>
    <xs:documentation>
      One of the values "yes" or "no" or "omit".
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:token">
    <xs:enumeration value="yes"/>
    <xs:enumeration value="no"/>
    <xs:enumeration value="omit"/>
  </xs:restriction>
</xs:simpleType>

</xs:schema>
```

H Acknowledgements (Non-Normative)

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 chair of the XSL WG is Sharon Adler, IBM. The XSL Working Group includes two overlapping teams working on XSLT and XSL Formatting Objects. The members of the XSL WG currently engaged in XSLT activities are:

Participant	Affiliation
Colin Paul Adams	Invited Expert
Anders Berglund	IBM
Scott Boag	IBM
Michael Kay	Invited Expert
Alex Milowski	Invited Expert
William Peterson	Novell, Inc
Michael Sperberg-McQueen	W3C
Zarella Rendon	Invited Expert
Jeni Tennison	Invited Expert
Joanne Tong	IBM
Norm Walsh	Sun Microsystems Inc.
Mohamed Zergaoui	Innovimax SARL
(vacancy)	Oracle

Alternates are listed only where they have taken an active part in working group discussions. However, the group acknowledges the support that many members receive from colleagues in their organizations, whether or not they are officially appointed as alternates.

The W3C representative on the XSL Working Group is Michael Sperberg-McQueen.

The following individuals made significant contributions to XSLT 2.0 while they were members of the Working Group, and in some cases afterwards:

James Clark, Invited Expert
Jonathan Marsh, Microsoft
Steve Muench, Oracle
Steve Zilles, Adobe
Evan Lenz, XYZFind
Mark Scardina, Oracle
Kristoffer Rose, IBM
Henry Zongaro, IBM
Henry Thompson, University of Edinburgh

K Karun, Oracle

The working group wishes to acknowledge the contribution made by David Marston of IBM especially to the new specification of the format-number function.

This specification builds on the success of the XSLT 1.0 Recommendation. For a list of contributors to XSLT 1.0, see [XSLT 1.0].

I Checklist of Requirements (Non-Normative)

This section provides a checklist of progress against the published XSLT 2.0 Requirements document (see [XSLT 2.0 Requirements]).

Requirement 1

MUST Maintain Backwards Compatibility with XSLT 1.1 [Read this as "with XSLT 1.0"]

Any stylesheet whose behavior is fully defined in XSLT 1.0 and which generates no errors will produce the same result tree under XSLT 2.0

Response

See *J.1 Incompatible Changes*

Requirement 2

MUST Match Elements with Null Values

A stylesheet should be able to match elements and attributes whose value is explicitly null.

Response

This has been handled as an XPath 2.0 requirement. A new function nilled FO is available to test whether an element has been marked as nil after schema validation.

Requirement 3

SHOULD Allow Included Documents to "Encapsulate" Local Stylesheets

XSLT 2.0 SHOULD define a mechanism to allow the templates in a stylesheet associated with a secondary source document, to be imported and used to format the included fragment, taking precedence over any applicable templates in the current stylesheet.

Response

The facility to define modes has been generalized, making it easier to define a distinct set of template rules for processing a particular document.

Requirement 4

Could Support Accessing Infoset Items for XML Declaration

A stylesheet COULD be able to access information like the version and encoding from the XML declaration of a document.

Response

No new facilities have been provided in this area, because this information is not available in the data model.

Requirement 5

Could Provide QName Aware String Functions

Users manipulating documents (for example stylesheets, schemas) that have QName-valued element or attribute content need functions that take a string containing a QName as their argument, convert it to an expanded-QName using either the namespace declarations in scope at that point in the stylesheet, or the namespace declarations in scope for a specific source node, and return properties of the expanded-QName such as its namespace URI and local name.

Response

Functions operating on QNames are included in the XPath 2.0 Functions and Operators document: see [Functions and Operators].

Requirement 6

Could Enable Constructing a Namespace with Computed Name

Provide an xsl:namespace analog to xsl:element for constructing a namespace node with a computed prefix and URI.

Response

An xsl:namespace instruction has been added: see *11.7 Creating Namespace Nodes*.

Requirement 7

Could Simplify Resolving Prefix Conflicts in QName-Valued Attributes

XSLT 2.0 could simplify the renaming of conflicting namespace prefixes in result tree fragments, particularly for attributes declared in a schema as being QNames. Once the processor knows an attribute value is a QName, an XSLT processor SHOULD be able to rename prefixes and generate namespace declarations to preserve the semantics of that attribute value, just as it does for attribute names.

Response

If an attribute is typed as a QName in the schema, the new XPath 2.0 functions can be used to manipulate it as required at application level. This is considered sufficient to meet the requirement.

Requirement 8

Could Support XHTML Output Method

Complementing the existing output methods for html, xml, and text, an xhtml output method could be provided to simplify transformations which target XHTML output.

Response

An XHTML output method is now provided: see [XSLT and XQuery Serialization]

Requirement 9

MUST Allow Matching on Default Namespace Without Explicit Prefix

Many users stumble trying to match an element with a default namespace.

Response

A new `[xsl:]xpath-default-namespace` attribute is provided for this purpose: see *5.2 Unprefixed QNames in Expressions and Patterns*

Requirement 10

MUST Add Date Formatting Functions

One of the more frequent requests from XSLT 1.0 users is the ability to format date information with similar control to XSLT's format-number. XML Schema introduces several kinds of date and time datatypes which will further increase the demand for date formatting during transformations. Functionality similar to that provided by java.text.SimpleDateFormat. A date analog of XSLT's named xsl:decimal-format may be required to handle locale-specific date formatting issues.

Response

A set of date formatting functions has been specified: see *16.5 Formatting Dates and Times*

Requirement 11

MUST Simplify Accessing Id's and Key's in Other Documents

Currently it is cumbersome to lookup nodes by id() or key() in documents other than the source document. Users MUST first use an xsl:for-each instruction, selecting the desired document() to make it the current node, then relative XPath expressions within the scope of the xsl:for-each can refer to id() or key() as desired.

Response

The requirement is met by the generalization of path syntax in XPath 2.0. It is now possible to use a path expression such as `document ('a.xml')/id('A001')`.

Requirement 12

SHOULD Provide Function to Absolutize Relative URIs

There SHOULD be a way in XSLT 2.0 to create an absolute URI. The functionality SHOULD allow passing a node-set and return a string value representing the absolute URI resolved with respect to the base URI of the current node.

Response

A function `resolve-uri`^{FO} is now defined in [Functions and Operators].

Requirement 13

SHOULD Include Unparsed Text from an External Resource

Frequently stylesheets MUST import text from external resources. Today users have to resort to extension functions to accomplish this because XSLT 1.0 only provides the document() function which, while useful, can only read external resources that are well-formed XML documents.

Response

A function `unparsed-text` has been added: see *16.2 Reading Text Files*

Requirement 14

SHOULD Allow Authoring Extension Functions in XSLT

As part of the XSLT 1.1 work done on extension functions, a proposal to author XSLT extension functions in XSLT itself was deferred for reconsideration in XSLT 2.0. This would allow the functions in an extension namespace to be implemented in "pure" XSLT, without resulting to external programming languages.

Response

A solution to this requirement, the `xsl:function` element, is included in this specification. See *10.3 Stylesheet Functions*.

Requirement 15

SHOULD Output Character Entity References Instead of Numeric Character Entities

Users have frequently requested the ability to have the output of their transformation use (named) character references instead of the numeric character entity. The ability to control this preference as the level of the whole document is sufficient. For example, rather than seeing   in the output, the user could request to see the equivalent   instead.

Response

The serialization specification gives the implementation discretion on how special characters are output. A user who wishes to force the use of named character references can achieve this using the new `xsl:character-map` declaration.

Requirement 16

SHOULD Construct Entity Reference by Name

Analogous to the ability to create elements and attributes, users have expressed a desire to construct named entity references.

Response

No solution has been provided to this requirement; it is difficult, because entity references are not defined in the data model.

Requirement 17

SHOULD Support for Unicode String Normalization

For reliable string comparison of Unicode strings, users need the ability to apply Unicode normalization before comparing the strings.

Response

This requirement has been addressed by the provision of the `normalize-unicode`[FO] function described in [Functions and Operators]. In addition, a serialization parameter `normalization-form` has been added.

Requirement 18

SHOULD Standardize Extension Element Language Bindings

XSLT 1.1 undertook the standardization of language bindings for XSLT extension functions. For XSLT 2.0, analogous bindings SHOULD be provided for extension elements [now renamed extension instructions].

Response

The XSL Working Group has decided not to pursue this requirement, and the attempt to standardize language bindings for extension functions that appeared in the XSLT 1.1 Working Draft has now been withdrawn. The Working Group decided that language bindings would be better published separately from the core XSLT specification.

Requirement 19

Could Improve Efficiency of Transformations on Large Documents

Many useful transformations take place on large documents consisting of thousands of repeating "sub-documents". Today transformations over these documents are impractical due to the need to have the entire source tree in memory. Enabling "progressive" transformations, where the processor is able to produce progressively more output as more input is received, is tantamount to avoiding the need for XSLT processors to have random access to the entire source document. This might be accomplished by:

Identifying a core subset of XPath that does not require random access to the source tree, or

Consider a "transform all subtrees" mode where the stylesheet says, "Apply the transformation implied by this stylesheet to each node that matches XXX, considered as the root of a separate tree, and copy all the results of these mini-transformations as separate subtrees on to the final result tree."

Response

The Working Group observes that implementation techniques for XSLT processing have advanced considerably since this requirement was written, and that further research developing new approaches continues both in industry and academia. In the light of these developments, the Working Group has decided that it would be inappropriate at this stage to identify language features or subsets designed specifically to enable progressive transformations.

Requirement 20

Could Support Reverse IDREF attributes

Given a particular value of an ID, produce a list of all elements that have an IDREF or IDREFS attribute which refers to this ID.

This functionality can be accomplished using the current <xsl:key> and key() mechanism.

Response

The `idref`[FO] function defined in [Functions and Operators] has been introduced in response to this requirement.

Requirement 21

Could Support Case-Insensitive Comparisons

XSLT 2.0 could expand its comparison functionality to include support for case-insensitive string comparison.

Response

This is an XPath 2.0 requirement. XPath 2.0 includes functions to convert strings to uppercase or lowercase, it also includes functions to compare strings using a named collating sequence, which provides the option of using a collating sequence that treats uppercase and lowercase as equal.

Requirement 22

Could Support Lexigraphic String Comparisons

We don't let users compare strings like $x > 'a'.

Response

This requirement has been addressed in XPath 2.0.

Requirement 23

Could Allow Comparing Nodes Based on Document Order

Support the ability to test whether one node comes before another in document order.

Response

This requirement has been addressed in XPath 2.0, using the operators << and >>.

Requirement 24

Could Improve Support for Unparsed Entities

In XSLT 1.0 there is an asymmetry in support for unparsed entities. They can be handled on input but not on output. In particular, there is no way to do an identity transformation that preserves them. At a minimum we need the ability to retrieve the Public ID of an unparsed entity.

Response

A function to retrieve the public identifier of an unparsed entity has been added. However, no facilities have been provided to include unparsed entities in a result document.

Requirement 25

Could Allow Processing a Node with the "Next Best Matching" Template

In the construction of large stylesheets for complex documents, it is often necessary to construct templates that implement special behavior for a particular instance of an element, and then apply the normal styling for that element. Currently this is not possible because `xsl:apply-templates` specifies that for any given node only a single template will be selected and instantiated.

Currently the processor determines a list of matching templates and then discards all but the one with the highest priority. In order to support this requirement, the processor would retain the list of matching templates sorted in priority order. A new instruction, for example `xsl:next-match`, in a template would simply trigger the next template in the list of matching templates. This "next best match" recursion naturally bottoms out at the builtin template which can be seen as the lowest priority matching template for every match pattern.

Response

An `xsl:next-match` instruction has been added.

Requirement 26

Could Make Coercions Symmetric By Allowing Scalar to Nodeset Conversion

Presently, no datatype can be coerced or cast to a node-set. By allowing a string value to convert to a node-set, some user "gotchas" could be avoided.

Response

The availability of sequences of strings or numbers probably meets most of the use-cases envisaged by this requirement.

Requirement 27

MUST Simplify Constructing and Copying Typed Content

It MUST be possible to construct XML Schema-typed elements and attributes. In addition, when copying an element or an attribute to the result, it SHOULD be possible to preserve the type during the process.

Response

Facilities to validate constructed and copied element and attribute nodes are defined in this specification; these elements and attributes will carry a type annotation indicating their XML Schema type. In addition, it is possible to specify when copying nodes whether type annotations should be preserved or removed.

Requirement 28

MUST Support Sorting Nodes Based on XML Schema Type

XSLT 1.0 supports sorting based on string-valued and number-valued expressions. XML Schema: Datatypes introduces new scalar types (for example, date) with well-known sort orders. It MUST be possible to sort based on these extended set of scalar data types. Since XML Schema: Datatypes does not define an ordering for complex types, this sorting support SHOULD only be considered for simple types.

SHOULD be consistent with whatever we define for the matrix of conversion and comparisons.

Response

Sorting based on any schema-defined primitive data type with a total ordering is included in this specification.

Requirement 29

Could Support Scientific Notation in Number Formatting

Several users have requested the ability to have the existing format-number() function extended to format numbers using Scientific Notation.

Response

Simple scientific formatting is now available through support for the schema-defined `xs:float` and `xs:double` data types; casting a large or small value of these types to a string produces a representation of the value in scientific notation. The Working Group believes that this will meet the requirement in most cases, and has therefore decided not to enhance the `format-number` further to introduce scientific notation. Users with more specialized requirements can write their own functions.

Requirement 30

Could Provide Ability to Detect Whether "Rich" Schema Information is Available

A stylesheet that requires XML Schema type-related functionality could be able to test whether a "rich" Post-Schema-Validated Infoset is available from the XML Schema processor, so that the stylesheet can provide fallback behavior or choose to exit with `xsl:message abort="yes"`.

Response

This requirement is satisified through the `instance of` operator in XPath 2.0, which allows expressions to determine the type of element and attribute nodes, using information from the schema. The details of how these expressions behave when there is no schema are defined in the XPath specifications.

Requirement 31

MUST Simplify Grouping

Grouping is complicated in XSLT 1.0. It MUST be possible for users to group nodes in a document based on common string-values, common names, or common values for any other expression

In addition XSLT MUST allow grouping based on sequential position, for example selecting groups of adjacent <P> elements. Ideally it SHOULD also make it easier to do fixed-size grouping as well, for example groups of three adjacent nodes, for laying out data in multiple columns. For each group of nodes identified, it MUST be possible to instantiate a template for the group. Grouping MUST be "nestable" to multiple levels so that groups of distinct nodes can be identified, then from among the distinct groups selected, further sub-grouping of distinct node in the current group can be done.

Response

A new `xsl:for-each-group` instruction is provided: see *14 Grouping*. In addition, many of the new functions and operators provided in XPath 2.0 make these algorithms easier to write.

J Changes from XSLT 1.0 (Non-Normative)

J.1 Incompatible Changes

This section lists all known cases where a stylesheet that was valid (produced no errors) under XSLT 1.0, and whose behavior was fully specified by XSLT 1.0, will produce different results under XSLT 2.0.

Most of the discussion is concerned with compatibility in the absence of a schema: that is, it is assumed that the source document being transformed has no schema when processed using XSLT 1.0, and that no schema is added when moving to XSLT 2.0. Some additional factors that come into play when a schema is added are noted at the end of the section.

J.1.1 Tree construction: whitespace stripping

Both in XSLT 1.0 and in XSLT 2.0, the XSLT specification places no constraints on the way in which source trees are constructed. For XSLT 2.0, however, the [Data Model] specification describes explicit processes for constructing a tree from an Infoset or a PSVI, while also permitting other processes to be used. The process described in [Data Model] has the effect of stripping whitespace text nodes from elements declared to have element-only content. Although the XSLT 1.0 specification did not preclude such behavior, it differs from the way that most existing XSLT 1.0 implementations work. It is RECOMMENDED that an XSLT 2.0 implementation wishing to provide maximum interoperability and backwards compatibility should offer the user the option either to construct source trees using the processes described in [Data Model], or alternatively to retain or remove whitespace according to the common practice of previous XSLT 1.0 implementations.

To write transformations that give the same result regardless of the whitespace stripping applied during tree construction, stylesheet authors can:

- use the `xsl:strip-space` declaration to remove whitespace text nodes from elements having element-only content (this has no effect if the whitespace has already been stripped)
- use instructions such as `<xsl:apply-templates select="*"/>` that cause only the element children of the context node to be

processed, and not its text nodes.

J.1.2 Changes in Serialization Behavior

The specification of the output of serialization is more prescriptive than in XSLT 1.0. For example, the `html` output method is REQUIRED to detect invalid HTML characters. Also, certain combinations of serialization parameters are now defined to be errors. Furthermore, XSLT 1.0 implementations were allowed to add additional `xsl:output` attributes that modified the behavior of the serializer. Some such extensions might be non-conformant under the stricter rules of XSLT 2.0. For example, some XSLT 1.0 processors provided an extension attribute to switch off the creation of `meta` elements by the `html` output method (a facility that is now provided as standard). A conformant XSLT 2.0 processor is not allowed to provide such extensions.

Where necessary, implementations MAY provide additional serialization methods designed to mimic more closely the behavior of specific XSLT 1.0 serializers.

J.1.3 Backwards Compatibility Behavior

Some XSLT constructs behave differently under XSLT 2.0 depending on whether backwards compatible behavior is enabled. In these cases, the behavior may be made compatible with XSLT 1.0 by ensuring that backwards compatible behavior is enabled (which is done using the `[xsl:]version` attribute).

These constructs are as follows:

1. If the `xsl:value-of` instruction has no `separator` attribute, and the value of the `select` expression is a sequence of more than one item, then under XSLT 2.0 all items in the sequence will be output, space separated, while in XSLT 1.0, all items after the first will be discarded.
2. If the effective value of an attribute value template is a sequence of more than one item, then under XSLT 2.0 all items in the sequence will be output, space separated, while in XSLT 1.0, all items after the first will be discarded.
3. If the expression in the `value` attribute of the `xsl:number` instruction returns a sequence of more than one item, then under XSLT 2.0 all items in the sequence will be output, as defined by the `format` attribute, but under XSLT 1.0, all items after the first will be discarded. If the sequence is empty, then under XSLT 2.0 nothing will be output (other than a prefix and suffix if requested), but under XSLT 1.0, the output is "NaN". If the first item in the sequence cannot be converted to a number, then XSLT 2.0 signals a non-recoverable error, while XSLT 1.0 outputs "NaN".

   If the expression in the `value` attribute of `xsl:number` returns an empty sequence or a sequence including non-numeric values, an XSLT 2.0 processor may signal a recoverable error; but with backwards compatibility enabled, it outputs `NaN`.
4. If the atomized value of the `select` attribute of the `xsl:sort` element is a sequence of more than one item, then under XSLT 2.0 an error will be signaled, while in XSLT 1.0, all items after the first will be discarded.
5. If an `xsl:call-template` instruction supplies a parameter that does not correspond to any template parameter in the template being called, then under XSLT 2.0 a static error is signaled, but under XSLT 1.0 the extra parameter is ignored.
6. It is normally a static error if an XPath expression contains a call to an unknown function. But when backwards compatible behavior is enabled, this is a non-recoverable dynamic error, which occurs only if the function call is actually evaluated.
7. An XSLT 1.0 processor compared the value of the expression in the `use` attribute of `xsl:key` to the value supplied in the second argument of the `key` function by converting both to strings. An XSLT 2.0 processor normally compares the values as supplied. The XSLT 1.0 behavior is retained if any of the `xsl:key` elements making up the key definition enables backwards-compatible behavior.
8. If no output method is explicitly requested, and the first element node output appears to be an XHTML document element, then under XSLT 2.0 the output method defaults to XHTML; with backwards compatibility enabled, the XML output method will be used.

Backwards compatible behavior also affects the results of certain XPath expressions, as defined in [XPath 2.0].

J.1.4 Incompatibility in the Absence of a Schema

If the source documents supplied as input to a transformation contain no type information generated from a schema then the known areas of incompatibility are as follows. These apply whether or not backwards compatible behavior is enabled.

1. A stylesheet that specifies a version number other than 1.0 was defined in XSLT 1.0 to execute in forwards-compatible mode; if such a stylesheet uses features that are not defined in XSLT 2.0 then errors may be signaled by an XSLT 2.0 processor that would not be signaled by an XSLT 1.0 processor.
2. At XSLT 1.0 the `system-property` function, when called with a first argument of `"xsl:version"`, returned 1.0 as a number. At XSLT 2.0 it returns "2.0" as a string. The RECOMMENDED way of testing this property is, for example, `<xsl:if test="number(system-property ('xsl:version')) &lt; 2.0">`, which will work with either an XSLT 1.0 or an XSLT 2.0 processor.
3. At XSLT 2.0 it is an error to specify the `mode` or `priority` attribute on an `xsl:template` element having no `match` attribute. At XSLT 1.0 the attributes were silently ignored in this situation.
4. When an `xsl:apply-templates` or `xsl:apply-imports` instruction causes a built-in template rule to be invoked, then any parameters that are supplied are automatically passed on to any further template rules. This did not happen in XSLT 1.0.
5. In XSLT 1.0 it was a recoverable error to create any node other than a text node while constructing the value of an attribute, comment, or processing-instruction; the recovery action was to ignore the offending node and its content. In XSLT 2.0 this is no longer an error, and the specified action is to atomize the node. An XSLT 2.0 processor will therefore not produce the same results as an XSLT 1.0 processor that took the error recovery action.
6. XSLT 1.0 defined a number of recoverable error conditions which in XSLT 2.0 have become non-recoverable errors. Under XSLT 1.0, a stylesheet that triggered such errors would fail under some XSLT processors and succeed (or at any rate, continue to completion) under others. Under XSLT 2.0 such a stylesheet will fail under all processors. Notable examples of such errors are constructing an element or attribute with an invalid name, generating attributes as children of a document node, and generating an attribute of an element after generating one or more children for the element. This change has been made in the interests of interoperability. In classifying such errors as non-recoverable, the Working Group used the criterion that no stylesheet author would be likely to write code that deliberately triggered the error and relied on the recovery action.
7. In XSLT 1.0, the semantics of tree construction were described as being top-down, in XSLT 2.0 they are described bottom up. In nearly all cases the end result is the same. One difference arises in the case of a tree that is constructed to contain an attribute node within a document node within an element node, using an instruction such as the following:

Example: Attribute within Document within Element

```
<xsl:template match="/">
  <e>
```

```
      <xsl:copy>
        <xsl:attribute name="a">5</xsl:attribute>
      </xsl:copy>
    </e>
  </xsl:template>
```

In XSLT 1.0, the xsl:copy did nothing, and the attribute a was then attached to the element e. In XSLT 2.0, an error occurs when attaching the attribute a to the document node constructed by xsl:copy, because this happens before the resulting document node is copied to the content of the constructed element.

8. In XSLT 1.0 it was not an error for the namespace attribute of xsl:element or xsl:attribute to evaluate to an invalid URI. Since many XML parsers accept any string as a namespace name, this rarely caused problems. The [Data Model], however, requires the name of a node to be an xs:QName, and the namespace part of an xs:QName is always an xs:anyURI. It is therefore now defined to be an error to create an element or attribute node in a namespace whose name is not a valid instance of xs:anyURI. In practice, however, implementations have some flexibility in how rigorously they validate namespace URIs.
9. It is now a static error for the stylesheet to contain two conflicting xsl:namespace-alias declarations with the same import precedence.
10. It is now a static error for an xsl:number instruction to contain both a value attribute and a level, from, or count attribute. In XSLT 1.0 the value attribute took precedence and the other attributes were silently ignored.
11. When the data-type attribute of xsl:sort has the value number, an XSLT 1.0 processor would evaluate the sort key as a string, and convert the result to a number. An XSLT 2.0 processor evaluates the sort key as a number directly. This only affects the outcome in cases where in XSLT 1.0, conversion of a number to a string and then back to a number does not produce the original number, as is the case for example with the number positive infinity.
12. When the data-type attribute of xsl:sort is omitted, an XSLT 1.0 processor would convert the sort key values to strings, and sort them as strings. An XSLT 2.0 processor will sort them according to their actual dynamic type. This means, for example, that if the sort key component specifies <xsl:sort select="string-length(.)"/>, an XSLT 2.0 processor will do a numeric sort where an XSLT 1.0 processor would have done an alphabetic sort.
13. When the data-type attribute of xsl:sort is omitted or has the value "text", an XSLT 1.0 processor treats a sort key whose value is an empty node-set as being equal to a sort key whose value is a zero-length string. XSLT 2.0 sorts the empty sequence before the zero-length string. This means that if there are two sort keys, say <xsl:sort select="@a"/> and <xsl:sort select="@b"/>, then an XSLT 1.0 processor will sort the element <x b="2"/> after <x a="" b="1"/>, while an XSLT 2.0 processor will produce the opposite ordering.
14. The specification of the format-number function has been rewritten to remove the normative dependency on the Java JDK 1.1 specification. The JDK 1.1 specification left aspects of the behavior undefined; it is therefore likely that some cases will give different results.

    The ability to include literal text in the format picture enclosed in single quotes has been removed; any stylesheet that uses this feature will need to be modified, for example to display the literal text using the concat[FO] function instead.

    One specific difference between the XSLT 2.0 specification and a JDK-based implementation is in the handling of the negative sub-picture. JDK releases subsequent to JDK 1.1 have added the provision: *If there is an explicit negative subpattern [sub-picture], it serves only to specify the negative prefix and suffix; the number of digits, minimal digits, and other characteristics are all the same as the positive pattern [sub-picture].* This statement was not present in the JDK 1.1 specification, and therefore it is not necessarily how every XSLT 1.0 implementation will behave, but it does describe the behavior of some XSLT 1.0 implementations that use the JDK directly. This behavior is not correct in XSLT 2.0: the negative sub-picture MUST be used as written when the number is negative.
15. The recovery action has changed for the error condition where the processor cannot handle the fragment identifier in a URI passed as an argument to the document function. XSLT 1.0 specified that the entire URI reference should be ignored. XSLT 2.0 specifies that the fragment identifier should be ignored.
16. XSLT 1.0 allowed the URI returned by the unparsed-entity-uri function to be derived from some combination of the system identifier and the public identifier in the source XML. XSLT 2.0 returns the system identifier as defined in the Infoset, resolved using the base URI of the source document. A new function is provided to return the public identifier.
17. The default priority of the pattern match="/" has changed from +0.5 to -0.5. The effect of this is that if there are any template rules that specify match="/" with an explicit user-specified priority between -0.5 and +0.5, these will now be chosen in preference to a template rule that specifies match="/" with no explicit priority; previously such rules would never have been invoked.
18. In XSLT 1.0 it was possible to create a processing instruction in the result tree whose string value contained a leading space. However, such leading spaces would be lost after serialization and parsing. In XSLT 2.0, any leading spaces in the string value of the processing instruction are removed at the time the node is created.
19. At XSLT 1.0 there were no restrictions on the namespaces that could be used for the names of user-defined stylesheet objects such as keys, variables, and named templates. In XSLT 2.0, certain namespaces (for example the XSLT namespace and the XML Schema namespace) are reserved.
20. An erratum to XSLT 1.0 specified what has become known as "sticky disable-output-escaping": specifically, that it should be possible to use disable-output-escaping when writing a node to a temporary tree, and that this information would be retained for use when the same node was later copied to a final result tree and serialized. XSLT 2.0 no longer specifies this behavior (though it permits it, at the discretion of the implementation). The use cases for this facility have been satisfied by a completely different mechanism, the concept of character maps (see *20.1 Character Maps*).

J.1.5 Compatibility in the Presence of a Schema

An XSLT 1.0 processor ignored all information about data types that might be obtained from a schema associated with a source document. An XSLT 2.0 processor will take account of such information, unless the input-type-annotations attribute is set to strip. This may lead to a number of differences in behavior. This section attempts only to give some examples of the kind of differences that might be expected when schema information is made available:

- Operations such as sorting will be sensitive to the data type of the items being sorted. For example, if the data type of a sort key component is defined in the schema as a date, then in the absence of a data-type attribute on the xsl:sort element, the sequence will be sorted in date order. With XSLT 1.0, the dates would be compared and sorted as strings.
- Certain operations that are permitted on untyped data are not permitted on typed data, if the type of the data is inappropriate for the operation. For example, the substring[FO] function expects its first argument to be a string. It is acceptable to supply an untyped value, which will be automatically converted to a string, but it is not acceptable to supply a value which has been annotated (as a result of schema processing) as an integer or a date.
- When an attribute value such as colors="red green blue" is processed without a schema, the value is considered to be a single string. When schema validation is applied, assuming the type is a list type like xs:NMTOKENS, the value will be treated as a sequence of three strings. This affects the results of many operations, for example comparison of the value with another string. With this attribute value, the expression contains(@colors, "green") returns true in XPath 1.0 and also in XPath 2.0 if input-type-annotations is set to strip. In XPath 2.0, with a schema-aware processor and with input-type-annotations set to preserve, the same expression returns false with backwards-compatibility enabled, and raises an error with backwards compatibility disabled.

J.1.6 XPath 2.0 Backwards Compatibility

Information about incompatibilities between XPath 2.0 and XPath 1.0 is included in [XPath 2.0]

Incompatibilities in the specification of individual functions in the core function library are listed in [Functions and Operators]

J.2 New Functionality

This section summarizes the new functionality offered in XSLT 2.0, compared with XSLT 1.0. These are arranged in three groups. Firstly, the changes that pervade the entire text. Secondly, the major new features introduced. And thirdly, a catalog of minor technical changes.

Changes since the November 2006 Proposed Recommendation are listed separately: see *J.2.4 Changes since Proposed Recommendation*.

In addition to these changes, reported errors in XSLT 1.0 have been fixed.

J.2.1 Pervasive changes

- There has been significant re-arrangement of the text. More terminology definitions have been hyperlinked, and a glossary (see *C Glossary*) has been added. Additional appendices summarize the error conditions and implementation-defined features of the specification.
- The specifications of many features (for example keys, `xsl:number`, the `format-number` function, the `xsl:import` mechanism, and the description of attribute sets) have been rewritten to make them clearer and more precise.
- Many changes have been made to support the XDM data model, notably the support for sequences as a replacement for the node-sets of XPath 1.0. This has affected the specification of elements such as `xsl:for-each`, `xsl:value-of`, and `xsl:sort`, and has led to the introduction of new instructions such as `xsl:sequence`.
- The processing model is described differently: instead of instructions "writing to the result tree", they now return sequences of values. This change is largely one of terminology, but it also means that it is now possible for XSLT stylesheets to manipulate arbitrary sequences, including sequences containing parentless element or attribute nodes.
- The description of the evaluation context has been changed. The concepts of current node and current node list have been replaced by the XPath concepts of context item, context position, and context size.
- With the introduction of support for XML Schema within XPath 2.0, XSLT now supports stronger data typing, while retaining backwards compatibility. In particular, the types of variables and parameters can now be specified explicitly, and schema validation can be invoked for result trees and for elements and attributes in temporary trees.
- The description of error handling has been improved (see *2.9 Error Handling*). This formalizes the difference between static and dynamic errors, and tightens the rules that define which errors must be signaled under which conditions.
- The terms implementation-defined and implementation-dependent are now defined and used consistently, and a checklist of implementation-defined features is provided (see *F Checklist of Implementation-Defined Features*).

J.2.2 Major Features

- XSLT 2.0 is designed to work with XPath 2.0 rather than XPath 1.0. This brings an enhanced data model with a type system based on sequences of nodes or atomic values, support for all the built-in types defined in XML Schema, and a wide range of new functions and operators.
- The result tree fragment data-type is eliminated. A variable-binding element with content (and no `as` attribute) now constructs a temporary tree, and the value of the variable is the root node of this tree (see *9.3 Values of Variables and Parameters*). With an `as` attribute, a variable-binding element may be used to construct an arbitrary sequence. These features eliminate the need for the `xx:node-set` extension function provided by many XSLT 1.0 implementations.
- Facilities are introduced for grouping of nodes (the `xsl:for-each-group` instruction, and the `current-group()` and `current-grouping-key()` functions). See *14 Grouping*
- It is now possible to create user-defined functions within the stylesheet, that can be called from XPath expressions. See *10.3 Stylesheet Functions*.
- A transformation is allowed to produce multiple result trees. See *19.1 Creating Final Result Trees*.
- A new instruction `xsl:analyze-string` is provided to process text by matching it against a regular expression.
- It is possible to declare the types of variables and parameters, and the result types of templates and functions. The types may either be built-in types, or user-defined types imported from a schema using a new `xsl:import-schema` declaration.
- A stylesheet is able to attach type annotations to elements and attributes in a result tree, and also in temporary trees, and to make use of any type annotations that exist in a source tree. Result trees and temporary trees can be validated against a schema.
- A transformation may now be invoked by calling a named template. This creates the potential for a transformation to process large collections of input documents. The input to such a transformation may be obtained using the `collection`^{FO} function defined in [Functions and Operators], or it may be supplied as a stylesheet parameter.
- Comparisons between values used for grouping, for sorting, and for keys can be performed using the rules for any supported data type, including the ability to select named collations for performing string comparison. These complement the new facilities in XPath 2.0, which are also invoked automatically when matching template rules.
- The `xsl:for-each` instruction is able to process any sequence, not only a sequence of nodes.
- An XHTML output method has been added. The details are described in [XSLT and XQuery Serialization].
- A `collation` attribute has been added to the `xsl:sort` element to allow sorting using a user-defined collation.
- A new `xsl:next-match` is provided to allow multiple template rules to be applied to the same source node.
- A new `xsl:character-map` declaration is available to control the serialization of individual characters. This is intended as a replacement for some use-cases where `disable-output-escaping` was previously necessary.
- Functions have been added for formatting dates and times. See *16.5 Formatting Dates and Times*
- The new facility of tunnel parameters allows parameters to be set that affect an entire phase of the transformation, without requiring them to be passed explicitly in every template call.
- Many instructions that previously constructed a value using child instructions can now alternatively construct the value using a `select` attribute; and conversely, instructions that previously required a `select` attribute can now use child instructions.
- The `xsl:template` declaration can now declare a template rule that applies to several different modes; and the `xsl:apply-templates` instruction can cause processing to continue in the current mode.

J.2.3 Minor Changes

- Instead of allowing the output method complete freedom to add namespace nodes, a process of namespace fixup is applied to the result tree before it is output; this same namespace fixup process is also applied to documents constructed using variable-binding elements with content (see *5.7.3 Namespace Fixup*).
- Support for XML Base has been added.
- An `xsl:apply-imports` element is allowed to have parameters (see *6.7 Overriding Template Rules* and *10.1.1 Passing Parameters to Templates*).
- Extension functions are allowed to return external objects, which do not have any of the builtin XPath types.
- The specification for patterns (*5.5 Patterns*) has been revised to align it with the new XPath grammar. The formal semantics of patterns has been simplified: this became possible because of the extra compositionality now available in the expression grammar. The syntax and semantics of patterns remains essentially unchanged, except that XPath 2.0 expressions can be used within predicates.
- A backwards-compatible processing mode is introduced. See *3.8 Backwards-Compatible Processing*
- The `system-property` function now always returns a string. Several new system properties have been defined. See *16.6.5 system-property*.
- With `<xsl:message terminate="yes">`, the processor now MUST terminate processing. Previously the word SHOULD was used. See *17 Messages*.
- A number of new serialization parameters have been introduced.
- A new instruction `xsl:namespace` is available, for creating namespace nodes: see *11.7 Creating Namespace Nodes*.
- A new instruction `xsl:perform-sort` is available, for returning a sorted sequence.
- A new `[xsl:]xpath-default-namespace` attribute is available to define the default namespace for unqualified names in an XPath expression or XSLT pattern.
- The attributes `[xsl:]version`, `[xsl:]exclude-result-prefixes`, and `[xsl:]extension-element-prefixes`, as well as the new `[xsl:]xpath-default-namespace` and `[xsl:]default-collation`, can be used on any XSLT element, not only on `xsl:stylesheet` and on literal result elements as before. In particular, they can now be used on the `xsl:template` element.
- A new `unparsed-text` function is introduced. It allows the contents of an external text file to be read as a string.
- Restrictions on the use of variables within patterns and key definitions have been removed; in their place a more general statement of the restrictions preventing circularity has been formulated. The `current` function may also now be used within patterns.
- The built-in templates for element and document nodes now pass any supplied parameter values on to the templates that they call.
- A detailed specification of the `format-number` function is now provided, removing the reliance on specifications in Java JDK 1.1.

J.2.4 Changes since Proposed Recommendation

The following changes have been made since publication of the Proposed Recommendation. Each change contains a reference to its discussion and rationale, for example the relevant issue number in the W3C public Bugzilla database.

- In *15.1 The xsl:analyze-string instruction*, the paragraph describing the permitted contents of the instruction has been clarified. (The sentence "Both elements are optional, and neither may appear more than once." was considered awkward). This editorial change was made in response to a public comment made during the Candidate Recommendation phase.
- In *19 Final Result Trees* it was stated that the result of a transformation consisted of zero or more result trees; while *2.4 Executing a Transformation* stated (correctly) that it consisted of one or more. The former statement has been revised. A cross-reference between the two sections has been added for clarification. (Bugzilla 4031)
- Some trivial syntax errors in examples have been fixed. (Bugzilla 4149)

The Proposed Recommendation contains a complete list of published working drafts prepared during the development of this specification, and a detailed history of changes may be assembled by viewing the change log present in each draft. For most of the drafts, a version is available in which changes are visually highlighted.