

XSL Transformations (XSLT) Version 3.0

W3C Recommendation 8 June 2017



This version:

<https://www.w3.org/TR/2017/REC-xslt-30-20170608/>

Latest version:

<https://www.w3.org/TR/xslt-30/>

Previous versions:

<https://www.w3.org/TR/2017/PR-xslt-30-20170418/>

<https://www.w3.org/TR/2017/CR-xslt-30-20170207/>

<https://www.w3.org/TR/2015/CR-xslt-30-20151119/>

<https://www.w3.org/TR/2014/WD-xslt-30-20141002/>

<https://www.w3.org/TR/2012/WD-xslt-30-20131212/>

<https://www.w3.org/TR/2012/WD-xslt-30-20120710/>

Editor:

Michael Kay, Saxonica <<http://www.saxonica.com/>>

Please check the **errata** for any errors or issues reported since publication.

See also **[translations](#)**.

The following associated resources are available:

- [Normative specification in HTML format](#)
- [HTML with revision markings \(non-normative\)](#)
- [XSD 1.1 Schema for XSLT 3.0 Stylesheets \(non-normative\)](#)
- [Relax-NG Schema for XSLT 3.0 Stylesheets \(non-normative\)](#)
- [XSD 1.0 Schema for the XML representation of JSON used by fn:json-to-xml \(non-normative\)](#)
- [Stylesheet for XML-to-JSON conversion \(non-normative\)](#)

[Copyright](#) © 2017 W3C® ([MIT](#), [ERCIM](#), [Keio](#), [Beihang](#)). W3C [liability](#), [trademark](#) and [document use](#) rules apply.

Abstract

This specification defines the syntax and semantics of XSLT 3.0, a language designed primarily for transforming XML documents into other XML documents.

XSLT 3.0 is a revised version of the XSLT 2.0 Recommendation [[XSLT 2.0](#)] published on 23 January 2007.

The primary purpose of the changes in this version of the language is to enable transformations to be performed in streaming mode, where neither the source document nor the result document is ever held in memory in its entirety. Another important aim is to improve the modularity of large stylesheets, allowing stylesheets to be developed from independently-developed components with a high level of software engineering robustness.

XSLT 3.0 is designed to be used in conjunction with XPath 3.0, which is defined in [\[XPath 3.0\]](#). XSLT shares the same data model as XPath 3.0, which is defined in [\[XDM 3.0\]](#), and it uses the library of functions and operators defined in [\[Functions and Operators 3.0\]](#). XPath 3.0 and the underlying function library introduce a number of enhancements, for example the availability of higher-order functions.

As an implementer option, XSLT 3.0 can also be used with XPath 3.1. All XSLT 3.0 processors provide maps, an addition to the data model which is specified (identically) in both XSLT 3.0 and XPath 3.1. Other features from XPath 3.1, such as arrays, and new functions such as [random-number-generator^{FO31}](#) and [sort^{FO31}](#), are available in XSLT 3.0 stylesheets only if the implementer chooses to support XPath 3.1.

Some of the functions that were previously defined in the XSLT 2.0 specification, such as the [format-date^{FO30}](#) and [format-number^{FO30}](#) functions, are now defined in the standard function library to make them available to other host languages.

XSLT 3.0 also includes optional facilities to serialize the results of a transformation, by means of an interface to the serialization component described in [\[XSLT and XQuery Serialization\]](#). Again, the new serialization capabilities of [\[XSLT and XQuery Serialization 3.1\]](#) are available at the implementer's option.

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 XP30 for XPath 3.0, DM30 for the XDM data model version 3.0, FO30 for Functions and Operators version 3.0.

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 <https://www.w3.org/TR/>.

This document is governed by the [1 March 2017 W3C Process Document](#).

This is a [Recommendation](#) of the W3C. It was developed by the W3C [XSLT Working Group](#).

This Recommendation specifies XSLT version 3.0. Changes since [XSLT 2.0](#) are listed in [J Changes since XSLT 2.0](#). The only incompatibilities with XSLT 2.0 relate to the way in which certain error conditions are handled: the details are given in [N Incompatibilities with XSLT 2.0](#).

No substantive changes have been made to this specification since its publication as a Proposed Recommendation. A few corrections and clarifications have been made to non-normative text: these are listed in [M Changes since the Proposed Recommendation of 18 April 2017](#).

This document has been reviewed by W3C Members, by software developers, and by other W3C groups and interested parties, and is endorsed by the Director as a W3C Recommendation. It is a stable document and may be used as reference material or cited 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 test suite for XSLT 3.0, containing over 11,000 test cases, is available at <https://dvcs.w3.org/hg/xslt30-test/>. The metadata for each test case describes any dependencies on optional or implementation-defined features of the specification, and provides expected results for each test. Documentation on how to run tests is available within the test suite. New tests may be added from time to time, and contributions are welcome.

An [implementation report](#) is available detailing test results for various implementations. This link points to the latest version of the report; older versions are available within the repository. New submissions of test results are welcome. Submitted test results and a stylesheet for generating the reports can be found within the repository.

This specification has been developed in conjunction with [\[XPath 3.0\]](#) and other documents that underpin both XSLT and XQuery. XSLT 3.0 REQUIRES support for XPath 3.0 augmented by a selection of features from XPath 3.1 which are described in [21 Maps](#) and [22 Processing JSON Data](#). XSLT 3.0 in addition allows a processor to support the whole of XPath 3.1, in which case it must do so as described in [27.7 XPath 3.1 Feature](#). In the event that future versions of XPath are defined beyond XPath 3.1, this specification allows XSLT 3.0 processors to provide support for such versions, but leaves it [implementation-defined](#) how this is done. References in this document to XPath and related specifications are by default to the 3.0 versions, but such references should be treated as version-agnostic unless the relevant prose indicates otherwise.

XSLT 3.0 specifies extensions to the XDM 3.0 data model, to the XPath 3.0 language syntax, and to the XPath 3.0 function library to underpin the introduction of maps, which were found necessary to support some XSLT streaming use cases, to enable XSLT to process JSON data, and to make many other processing tasks easier. These extensions have been incorporated into XDM 3.1 and XPath 3.1. Although XDM 3.1 and XPath 3.1 have reached Recommendation status, XSLT 3.0 has not been made dependent on XPath 3.1, other than those features needed to meet the XSLT 3.0 requirements.

Please report errors in this document using W3C's [public Bugzilla system](#) (instructions can be found at <https://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 will be very helpful if you include the string “[XSLT30]” in the subject line of your report, whether made in Bugzilla or in email. Please use multiple Bugzilla entries (or, if necessary, multiple email messages) if you have more than one comment to make. Archives of the comments and responses are available at <https://lists.w3.org/Archives/Public/public-qt-comments/>.

The same mechanism may be used for reporting errors in the test suite.

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?
1.2	What's New in XSLT 3.0?
2	Concepts
2.1	Terminology
2.2	Notation
2.3	Initiating a Transformation
2.3.1	Information needed for Static Analysis
2.3.2	Priming a Stylesheet
2.3.3	Apply-Templates Invocation

- 2.3.4 Call-Template Invocation
- 2.3.5 Function Call Invocation
- 2.3.6 Post-processing the Raw Result
- 2.3.6.1 Result Tree Construction
- 2.3.6.2 Serializing the Result
- 2.4 Instructions
- 2.5 Rule-Based Processing
- 2.6 The Evaluation Context
- 2.7 Parsing and Serialization
- 2.8 Packages and Modules
- 2.9 Extensibility
- 2.10 Stylesheets and XML Schemas
- 2.11 Streaming
- 2.12 Streamed Validation
- 2.13 Streaming of non-XML data
- 2.14 Error Handling

3 Stylesheet Structure

- 3.1 XSLT Namespace
- 3.2 Extension Attributes
- 3.3 XSLT Media Type
- 3.4 Standard Attributes
- 3.5 Packages
 - 3.5.1 Versions of a Package
 - 3.5.2 Dependencies between Packages
 - 3.5.3 Named Components in Packages
 - 3.5.3.1 Visibility of Components
 - 3.5.3.2 Accepting Components
 - 3.5.3.3 Overriding Components from a Used Package
 - 3.5.3.4 Referring to Overridden Components
 - 3.5.3.5 Binding References to Components
 - 3.5.3.6 Dynamic References to Components
 - 3.5.4 Overriding Template Rules from a Used Package
 - 3.5.4.1 Requiring Explicit Mode Declarations
 - 3.5.5 Declarations Local to a Package
 - 3.5.6 Declaring the Global Context Item
 - 3.5.7 Worked Example of a Library Package
 - 3.5.7.1 Default Functionality of the CSV Package
 - 3.5.7.2 Package Structure
 - 3.5.7.3 The csv:parse Function and its User-customization Hooks
 - 3.5.7.4 Breaking the Input into Lines
 - 3.5.7.5 Pre-processing the Lines
 - 3.5.7.6 The Mode csv:parse-line
 - 3.5.7.7 Mode csv:parse-field
 - 3.5.7.8 The csv:quote Variable
 - 3.5.7.9 The csv:preprocess-field Function
 - 3.5.7.10 The Mode csv:post-process
 - 3.5.7.11 Overriding the Default Behavior
 - 3.6 Stylesheet Modules
 - 3.7 Stylesheet Element
 - 3.7.1 The default-collation Attribute

- 3.7.2 The default-mode Attribute
- 3.7.3 User-defined Data Elements
- 3.8 Simplified Stylesheet Modules
- 3.9 Backwards Compatible Processing
 - 3.9.1 XSLT 1.0 Compatibility Mode
 - 3.9.2 XSLT 2.0 Compatibility Mode
- 3.10 Forwards Compatible Processing
- 3.11 Combining Stylesheet Modules
 - 3.11.1 Locating Stylesheet Modules
 - 3.11.2 Stylesheet Inclusion
 - 3.11.3 Stylesheet Import
- 3.12 Embedded Stylesheet Modules
- 3.13 Stylesheet Preprocessing
 - 3.13.1 Conditional Element Inclusion
 - 3.13.2 Shadow Attributes
- 3.14 Built-in Types
- 3.15 Importing Schema Components

4 Data Model

- 4.1 XML Versions
- 4.2 XDM versions
- 4.3 Stripping Whitespace from the Stylesheet
- 4.4 Preprocessing Source Documents
 - 4.4.1 Stripping Type Annotations from a Source Tree
 - 4.4.2 Stripping Whitespace from a Source Tree
- 4.5 Attribute Types and DTD Validation
- 4.6 Data Model for Streaming
 - 4.6.1 Streamed Documents
 - 4.6.2 Other Data Structures
- 4.7 Limits
- 4.8 Disable Output Escaping

5 Features of the XSLT Language

- 5.1 Names
 - 5.1.1 Qualified Names
 - 5.1.2 Unprefixed Lexical QNames in Expressions and Patterns
 - 5.1.3 Reserved Namespaces
- 5.2 Expressions
- 5.3 The Static and Dynamic Context
 - 5.3.1 Initializing the Static Context
 - 5.3.2 Additional Static Context Components used by XSLT
 - 5.3.3 Initializing the Dynamic Context
 - 5.3.3.1 Maintaining Position: the Focus
 - 5.3.3.2 Other Components of the XPath Dynamic Context
 - 5.3.4 Additional Dynamic Context Components used by XSLT
- 5.4 Defining a Decimal Format
- 5.5 Patterns
 - 5.5.1 Examples of Patterns

5.5.2	Syntax of Patterns
5.5.3	The Meaning of a Pattern
5.5.4	Errors in Patterns
5.6	Value Templates
5.6.1	Attribute Value Templates
5.6.2	Text Value Templates
5.7	Sequence Constructors
5.7.1	Constructing Complex Content
5.7.2	Constructing Simple Content
5.7.3	Namespace Fixup
5.8	URI References

6 Template Rules

6.1	Defining Templates
6.2	Defining Template Rules
6.3	Applying Template Rules
6.4	Conflict Resolution for Template Rules
6.5	Default Priority for Template Rules
6.6	Modes
6.6.1	Declaring Modes
6.6.2	Using Modes
6.6.3	Declaring the Type of Nodes Processed by a Mode
6.6.4	Streamable Templates
6.7	Built-in Template Rules
6.7.1	Built-in Templates: Text-only Copy
6.7.2	Built-in Templates: Deep Copy
6.7.3	Built-in Templates: Shallow Copy
6.7.4	Built-in Templates: Deep Skip
6.7.5	Built-in Templates: Shallow Skip
6.7.6	Built-in Templates: Fail
6.8	Overriding Template Rules
6.9	Passing Parameters to Template Rules

7 Repetition

7.1	The <code>xsl:for-each</code> instruction
7.2	The <code>xsl:iterate</code> Instruction

8 Conditional Processing

8.1	Conditional Processing with <code>xsl:if</code>
8.2	Conditional Processing with <code>xsl:choose</code>
8.3	Try/Catch
8.3.1	Recovery of Result Trees
8.3.2	Try/Catch Examples
8.4	Conditional Content Construction
8.4.1	The <code>xsl:where-populated</code> instruction
8.4.2	The <code>xsl:on-empty</code> instruction
8.4.3	The <code>xsl:on-non-empty</code> instruction
8.4.4	Evaluating <code>xsl:on-empty</code> and <code>xsl:on-non-empty</code> Instructions

8.4.5 A More Complex Example

9 Variables and Parameters

9.1 Variables

9.2 Parameters

9.2.1 The Required Type of a Parameter

9.2.2 Default Values of Parameters

9.3 Values of Variables and Parameters

9.4 Creating Implicit Document Nodes

9.5 Global Variables and Parameters

9.6 Static Variables and Parameters

9.7 Static Expressions

9.8 Local Variables and Parameters

9.9 Scope of Variables

9.10 Setting Parameter Values

9.11 Circular Definitions

10 Callable Components

10.1 Named Templates

10.1.1 Declaring the Context Item for a Template

10.1.2 Passing Parameters to Named Templates

10.1.3 Tunnel Parameters

10.2 Named Attribute Sets

10.2.1 Using Attribute Sets

10.2.2 Visibility of Attribute Sets

10.2.3 Streamability of Attribute Sets

10.2.4 Evaluating Attribute Sets

10.2.5 Attribute Sets: Examples

10.3 Stylesheet Functions

10.3.1 Function Name and Arity

10.3.2 Arguments

10.3.3 Function Result

10.3.4 Visibility and Overriding of Functions

10.3.5 Streamability of Stylesheet Functions

10.3.6 Dynamic Access to Functions

10.3.7 Determinism of Functions

10.3.8 Memoization

10.3.9 Examples of Stylesheet Functions

10.4 Dynamic XPath Evaluation

10.4.1 Static context for the target expression

10.4.2 Dynamic context for the target expression

10.4.3 The effect of the xsl:evaluate instruction

10.4.4 xsl:evaluate as an optional feature

10.4.5 Examples of xsl:evaluate

11 Creating Nodes and Sequences

11.1 Literal Result Elements

11.1.1 Setting the Type Annotation for Literal Result Elements

- 11.1.2 Attribute Nodes for Literal Result Elements
- 11.1.3 Namespace Nodes for Literal Result Elements
- 11.1.4 Namespace Aliasing
- 11.2 Creating Element Nodes Using `xsl:element`
 - 11.2.1 The Content of the Constructed Element Node
 - 11.2.2 The Name of the Constructed Element Node
 - 11.2.3 Other Properties of the Constructed Element Node
 - 11.2.4 The Type Annotation of the Constructed Element Node
- 11.3 Creating Attribute Nodes Using `xsl:attribute`
 - 11.3.1 Setting the Type Annotation for a Constructed Attribute Node
- 11.4 Creating Text Nodes
 - 11.4.1 Literal Text Nodes
 - 11.4.2 Creating Text Nodes Using `xsl:text`
 - 11.4.3 Generating Text with `xsl:value-of`
- 11.5 Creating Document Nodes
- 11.6 Creating Processing Instructions
- 11.7 Creating Namespace Nodes
- 11.8 Creating Comments
- 11.9 Copying Nodes
 - 11.9.1 Shallow Copy
 - 11.9.2 Deep Copy
- 11.10 Constructing Sequences

12 Numbering

- 12.1 The `start-at` Attribute
- 12.2 Formatting a Supplied Number
- 12.3 Numbering based on Position in a Document
- 12.4 Number to String Conversion Attributes

13 Sorting

- 13.1 The `xsl:sort` Element
 - 13.1.1 The Sorting Process
 - 13.1.2 Comparing Sort Key Values
 - 13.1.3 Sorting Using Collations
- 13.2 Creating a Sorted Sequence
- 13.3 Processing a Sequence in Sorted Order
- 13.4 The Unicode Collation Algorithm

14 Grouping

- 14.1 The `xsl:for-each-group` Element
- 14.2 Accessing Information about the Current Group Value
 - 14.2.1 `fn:current-group`
 - 14.2.2 `fn:current-grouping-key`
- 14.3 Ordering among Groups
- 14.4 Examples of Grouping
- 14.5 Non-Transitivity

15 Merging

- 15.1 Terminology for Merging
- 15.2 The xsl:merge Instruction
- 15.3 Selecting the Sequences to be Merged
- 15.4 Streamable Merging
- 15.5 Defining the Merge Keys
- 15.6 The Current Merge Group and Key
 - 15.6.1 fn:current-merge-group
 - 15.6.2 fn:current-merge-key
- 15.7 The xsl:merge-action Element
- 15.8 Examples of xsl:merge

16 Splitting

- 16.1 The xsl:fork Instruction
- 16.2 Examples of Splitting with Streamed Data

17 Regular Expressions

- 17.1 The xsl:analyze-string Instruction
- 17.2 fn:regex-group
- 17.3 Examples of Regular Expression Matching

18 Streaming

- 18.1 The xsl:source-document Instruction
 - 18.1.1 Validation of Source Documents
 - 18.1.2 Examples of xsl:source-document
 - 18.1.3 fn:stream-available
- 18.2 Accumulators
 - 18.2.1 Declaring an Accumulator
 - 18.2.2 Applicability of Accumulators
 - 18.2.3 Informal Model for Accumulators
 - 18.2.4 Formal Model for Accumulators
 - 18.2.5 Dynamic Errors in Accumulators
 - 18.2.6 fn:accumulator-before
 - 18.2.7 fn:accumulator-after
 - 18.2.8 Importing of Accumulators
 - 18.2.9 Streamability of Accumulators
 - 18.2.10 Copying Accumulator Values
 - 18.2.11 Examples of Accumulators
- 18.3 fn:copy-of
- 18.4 fn:snapshot

19 Streamability

- 19.1 Determining the Static Type of a Construct
 - 19.1.1 Static Type of an Axis Step
 - 19.1.2 Static Type of a Call to current
 - 19.1.3 Schema-Aware Streamability Analysis
- 19.2 Determining the Context Item Type
- 19.3 Operand Roles
 - 19.3.1 Examples showing the Effect of Operand Usage

- 19.4 Determining the Posture of a Construct
- 19.5 Determining the Context Posture
- 19.6 The Sweep of a Construct
- 19.7 Grounded Consuming Constructs
- 19.8 Classifying Constructs
 - 19.8.1 General Rules for Streamability
 - 19.8.2 Examples of the General Streamability Rules
 - 19.8.3 Classifying Sequence Constructors
 - 19.8.4 Classifying Instructions
 - 19.8.4.1 Streamability of Literal Result Elements
 - 19.8.4.2 Streamability of extension instructions
 - 19.8.4.3 Streamability of `xsl:analyze-string`
 - 19.8.4.4 Streamability of `xsl:apply-imports`
 - 19.8.4.5 Streamability of `xsl:apply-templates`
 - 19.8.4.6 Streamability of `xsl:assert`
 - 19.8.4.7 Streamability of `xsl:attribute`
 - 19.8.4.8 Streamability of `xsl:break`
 - 19.8.4.9 Streamability of `xsl:call-template`
 - 19.8.4.10 Streamability of `xsl:choose`
 - 19.8.4.11 Streamability of `xsl:comment`
 - 19.8.4.12 Streamability of `xsl:copy`
 - 19.8.4.13 Streamability of `xsl:copy-of`
 - 19.8.4.14 Streamability of `xsl:document`
 - 19.8.4.15 Streamability of `xsl:element`
 - 19.8.4.16 Streamability of `xsl:evaluate`
 - 19.8.4.17 Streamability of `xsl:fallback`
 - 19.8.4.18 Streamability of `xsl:for-each`
 - 19.8.4.19 Streamability of `xsl:for-each-group`
 - 19.8.4.20 Streamability of `xsl:fork`
 - 19.8.4.21 Streamability of `xsl:if`
 - 19.8.4.22 Streamability of `xsl:iterate`
 - 19.8.4.23 Streamability of `xsl:map`
 - 19.8.4.24 Streamability of `xsl:map-entry`
 - 19.8.4.25 Streamability of `xsl:merge`
 - 19.8.4.26 Streamability of `xsl:message`
 - 19.8.4.27 Streamability of `xsl:namespace`
 - 19.8.4.28 Streamability of `xsl:next-iteration`
 - 19.8.4.29 Streamability of `xsl:next-match`
 - 19.8.4.30 Streamability of `xsl:number`
 - 19.8.4.31 Streamability of `xsl:on-empty`
 - 19.8.4.32 Streamability of `xsl:on-non-empty`
 - 19.8.4.33 Streamability of `xsl:perform-sort`
 - 19.8.4.34 Streamability of `xsl:processing-instruction`
 - 19.8.4.35 Streamability of `xsl:result-document`
 - 19.8.4.36 Streamability of `xsl:sequence`
 - 19.8.4.37 Streamability of `xsl:source-document`
 - 19.8.4.38 Streamability of `xsl:text`
 - 19.8.4.39 Streamability of `xsl:try`
 - 19.8.4.40 Streamability of `xsl:value-of`
 - 19.8.4.41 Streamability of `xsl:variable`
 - 19.8.4.42 Streamability of `xsl:where-populated`
 - 19.8.5 Classifying Stylesheet Functions
 - 19.8.5.1 Streamability Category: unclassified
 - 19.8.5.2 Streamability Category: absorbing
 - 19.8.5.3 Streamability Category: inspection
 - 19.8.5.4 Streamability Category: filter

- 19.8.5.5 Streamability Category: shallow-descent
- 19.8.5.6 Streamability Category: deep-descent
- 19.8.5.7 Streamability Category: ascent
- 19.8.6 Classifying Attribute Sets
- 19.8.7 Classifying Value Templates
- 19.8.8 Classifying Expressions
- 19.8.8.1 Streamability of Expressions
- 19.8.8.2 Streamability of Quantified Expressions
- 19.8.8.3 Streamability of if expressions
- 19.8.8.4 Streamability of union, intersect, and except Expressions
- 19.8.8.5 Streamability of instance of Expressions
- 19.8.8.6 Streamability of treat as Expressions
- 19.8.8.7 Streamability of Simple Mapping Expressions
- 19.8.8.8 Streamability of Path Expressions
- 19.8.8.9 Streamability of Axis Steps
- 19.8.8.10 Streamability of Filter Expressions
- 19.8.8.11 Streamability of Dynamic Function Calls
- 19.8.8.12 Streamability of Variable References
- 19.8.8.13 Streamability of the Context Item Expression
- 19.8.8.14 Streamability of Static Function Calls
- 19.8.8.15 Streamability of Named Function References
- 19.8.8.16 Streamability of Inline Function Declarations
- 19.8.8.17 Streamability of Map Constructors
- 19.8.8.18 Streamability of Lookup Expressions
- 19.8.9 Classifying Calls to Built-In Functions
- 19.8.9.1 Streamability of the accumulator-after Function
- 19.8.9.2 Streamability of the accumulator-before Function
- 19.8.9.3 Streamability of the current Function
- 19.8.9.4 Streamability of the current-group Function
- 19.8.9.5 Streamability of the current-grouping-key Function
- 19.8.9.6 Streamability of the current-merge-group Function
- 19.8.9.7 Streamability of the current-merge-key Function
- 19.8.9.8 Streamability of the fold-left Function
- 19.8.9.9 Streamability of the fold-right Function
- 19.8.9.10 Streamability of the for-each Function
- 19.8.9.11 Streamability of the for-each-pair Function
- 19.8.9.12 Streamability of the function-lookup Function
- 19.8.9.13 Streamability of the innermost Function
- 19.8.9.14 Streamability of the last Function
- 19.8.9.15 Streamability of the outermost Function
- 19.8.9.16 Streamability of the position Function
- 19.8.9.17 Streamability of the reverse Function
- 19.8.9.18 Streamability of the root Function
- 19.8.10 Classifying Patterns
- 19.9 Examples of Streamability Analysis
- 19.10 Streamability Guarantees

20 Additional Functions

- 20.1 fn:document
- 20.2 Keys
 - 20.2.1 The xsl:key Declaration
 - 20.2.2 fn:key
- 20.3 Keys and Streaming
- 20.4 Miscellaneous Additional Functions
 - 20.4.1 fn:current

- 20.4.2 fn:unparsed-entity-uri
- 20.4.3 fn:unparsed-entity-public-id
- 20.4.4 fn:system-property
- 20.4.5 fn:available-system-properties

21 Maps

- 21.1 The Type of a Map
- 21.2 Functions that Operate on Maps
 - 21.2.1 op:same-key
 - 21.2.2 map:merge
 - 21.2.3 map:size
 - 21.2.4 map:keys
 - 21.2.5 map:contains
 - 21.2.6 map:get
 - 21.2.7 map:put
 - 21.2.8 map:entry
 - 21.2.9 map:remove
 - 21.2.10 map:for-each
 - 21.2.11 map:find
 - 21.2.12 fn:collation-key
 - 21.2.13 fn:deep-equal
- 21.3 Map Instructions
- 21.4 Map Constructors
- 21.5 The Map Lookup Operator
 - 21.5.1 The Unary Lookup Operator
 - 21.5.2 The Postfix Lookup Operator
- 21.6 Maps and Streaming
- 21.7 Examples using Maps

22 Processing JSON Data

- 22.1 XML Representation of JSON
- 22.2 Option Parameter Conventions
- 22.3 fn:json-to-xml
- 22.4 fn:xml-to-json
- 22.5 Transforming XML to JSON

23 Diagnostics

- 23.1 Messages
- 23.2 Assertions

24 Extensibility and Fallback

- 24.1 Extension Functions
 - 24.1.1 fn:function-available
 - 24.1.2 Calling Extension Functions
 - 24.1.3 External Objects
 - 24.1.4 fn:type-available
- 24.2 Extension Instructions
 - 24.2.1 Designating an Extension Namespace

24.2.2 fn:element-available

24.2.3 Fallback

25 Transformation Results

25.1 Creating Secondary Results

25.2 Restrictions on the use of xsl:result-document

25.3 The Current Output URI

25.3.1 fn:current-output-uri

25.4 Validation

25.4.1 Validating Constructed Elements and Attributes

25.4.1.1 Validation using the [xsl:]validation Attribute

25.4.1.2 Validation using the [xsl:]type Attribute

25.4.1.3 The Validation Process

25.4.2 Validating Document Nodes

25.4.3 Validating xml:id attributes

26 Serialization

26.1 Character Maps

26.2 Disabling Output Escaping

27 Conformance

27.1 Basic XSLT Processor

27.2 Schema-Awareness Conformance Feature

27.3 Serialization Feature

27.4 Compatibility Features

27.5 Streaming Feature

27.6 Dynamic Evaluation Feature

27.7 XPath 3.1 Feature

27.7.1 Arrays

27.8 Higher-Order Functions Feature

A References

A.1 Normative References

A.2 Other References

B XML Representation of JSON

B.1 Schema for the XML Representation of JSON

B.2 Stylesheet for converting XML to JSON

C Glossary (Non-Normative)

D Element Syntax Summary (Non-Normative)

E Summary of Error Conditions (Non-Normative)

F Checklist of Implementation-Defined Features (Non-Normative)

F.1 Application Programming Interfaces

- F.2 Vendor and User Extensions
- F.3 Localization
- F.4 Optional Features
- F.5 Dependencies
- F.6 Defaults and Limits
- F.7 Detection and Reporting of Errors

**G Summary of Available Functions
(Non-Normative)**

- G.1 Function Classification
- G.2 List of XSLT-defined functions

**H Schemas for XSLT 3.0 Stylesheets
(Non-Normative)**

- H.1 XSD 1.1 Schema for XSLT Stylesheets
- H.2 Relax-NG Schema for XSLT Stylesheets

**I Acknowledgements
(Non-Normative)**

**J Changes since XSLT 2.0
(Non-Normative)**

- J.1 Changes in this Specification
- J.2 Changes in Other Related Specifications

**K Changes since the Candidate Recommendation of 19 November 2015
(Non-Normative)**

**L Changes since the Candidate Recommendation of 7 February 2017
(Non-Normative)**

**M Changes since the Proposed Recommendation of 18 April 2017
(Non-Normative)**

**N Incompatibilities with XSLT 2.0
(Non-Normative)**

❖ ❖ ❖

1 Introduction

1.1 What is XSLT?

This specification defines the syntax and semantics of the XSLT 3.0 language.

A transformation in the XSLT language is expressed in the form of a **stylesheet**. A stylesheet is made up of one or more well-formed XML [[XML 1.0](#)] documents conforming to the Namespaces in XML Recommendation [[Namespaces in XML](#)].

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 [\[XSL-FO\]](#)), 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 input data into output data. The inputs and outputs will all be instances of the XDM data model, described in [\[XDM 3.0\]](#). In the simplest and most common case, the input is an XML document referred to as the source tree, and the output is an XML document referred to as the result tree. It is also possible to process multiple source documents, to generate multiple result documents, and to handle formats other than XML. The transformation is achieved by a set of [template rules](#). A template rule associates a [pattern](#), which typically 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.

Stylesheets have a modular structure; they may contain several packages developed independently of each other, and each package may consist of several stylesheet modules.

[**DEFINITION:** A [stylesheet](#) consists of one or more packages: specifically, one [top-level package](#) and zero or more [library packages](#).]

[**DEFINITION:** For a given transformation, one [package](#) functions as the **top-level package**. The complete [stylesheet](#) is assembled by finding the packages referenced directly or indirectly from the top-level package using [xsl:use-package](#) declarations: see [3.5.2 Dependencies between Packages](#).]

[**DEFINITION:** Every [package](#) within a [stylesheet](#), other than the [top-level package](#), is referred to as a **library package**.]

[**DEFINITION:** Within a [package](#), one [stylesheet module](#) functions as the **principal stylesheet module**. The complete package is assembled by finding the stylesheet modules referenced directly or indirectly from the principal stylesheet module using [xsl:include](#) and [xsl:import](#) elements: see [3.11.2 Stylesheet Inclusion](#) and [3.11.3 Stylesheet Import](#).]

[1.2 What's New in XSLT 3.0?](#)

A major focus for enhancements in XSLT 3.0 is the requirement to enable streaming of source documents. This is needed when source documents become too large to hold in main memory, and also for applications where it is important to start delivering results before the entire source document is available.

While implementations of XSLT that use streaming have always been theoretically possible, the nature of the language has made it very difficult to achieve this in practice. The approach adopted in this specification is twofold: it identifies a set of restrictions which, if followed by stylesheet authors, will enable implementations to adopt a

streaming mode of operation without placing excessive demands on the optimization capabilities of the processor; and it provides new constructs to indicate that streaming is required, or to express transformations in a way that makes it easier for the processor to adopt a streaming execution plan.

Capabilities provided in this category include:

- A new [`xsl:source-document`](#) instruction, which reads and processes a source document, optionally in streaming mode;
- The ability to declare that a [`mode`](#) is a streaming mode, in which case all the template rules using that mode must be streamable;
- A new [`xsl:iterate`](#) instruction, which iterates over the items in a sequence, allowing parameters for the processing of one item to be set during the processing of the previous item;
- A new [`xsl:merge`](#) instruction, allowing multiple input streams to be merged into a single output stream;
- A new [`xsl:fork`](#) instruction, allowing multiple computations to be performed in parallel during a single pass through an input document;
- [Accumulators](#), which allow a value to be computed progressively during streamed processing of a document, and accessed as a function of a node in the document, without compromise to the functional nature of the XSLT language.

A second focus for enhancements in XSLT 3.0 is the introduction of a new mechanism for stylesheet modularity, called the package. Unlike the stylesheet modules of XSLT 1.0 and 2.0 (which remain available), a package defines an interface that regulates which functions, variables, templates and other components are visible outside the package, and which can be overridden. There are two main goals for this facility: it is designed to deliver software engineering benefits by improving the reusability and maintainability of code, and it is intended to streamline stylesheet deployment by allowing packages to be compiled independently of each other, and compiled instances of packages to be shared between multiple applications.

Other significant features in XSLT 3.0 include:

- An [`xsl:evaluate`](#) instruction allowing evaluation of XPath expressions that are dynamically constructed as strings, or that are read from a source document;
- Enhancements to the syntax of [`patterns`](#), in particular enabling the matching of atomic values as well as nodes;
- An [`xsl:try`](#) instruction to allow recovery from dynamic errors;
- The element [`xsl:global-context-item`](#), used to declare the stylesheet's expectations of the global context item (notably, its type);
- A new instruction [`xsl:assert`](#) to assist developers in producing correct and robust code.

XSLT 3.0 also delivers enhancements made to the XPath language and to the standard function library, including the following:

- Variables can now be bound in XPath using the `let` expression.
- Functions are now first class values, and can be passed as arguments to other (higher-order) functions, making XSLT a fully-fledged functional programming language.
- A number of new functions are available, for example trigonometric functions, and the functions [`parse-xml`](#)^{FO30} and [`serialize`](#)^{FO30} to convert between lexical and tree representations of XML.

XSLT 3.0 also includes support for maps (a data structure consisting of key/value pairs, sometimes referred to in other programming languages as dictionaries, hashes, or associative arrays). This feature extends the data model, provides new syntax in XPath, and adds a number of new functions and operators. Initially developed as XSLT-specific extensions, maps have now been integrated into XPath 3.1 (see [\[XPath 3.1\]](#)). XSLT 3.0 does not require implementations to support XPath 3.1 in its entirety, but it does require support for these specific features.

A full list of changes is at [J Changes since XSLT 2.0](#).

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 **tree** is used (as in [\[XDM 3.0\]](#)) to refer to the aggregate consisting of a parentless node together with all its descendant nodes, plus all their attributes and namespaces.]

Note:

The use of the term **tree** in this document does not imply the use of a data structure in memory that holds the entire contents of the document at one time. It implies rather a logical view of the XML input and output in which elements have a hierarchic relationship to each other. When a source document is being processed in a streaming manner, access to the nodes in this tree is constrained, but it is still viewed and described as a tree.

The output of a transformation consists of the following:

1. [**DEFINITION:** A **principal result**: this can be any sequence of items (as defined in [\[XDM 3.0\]](#)).] The principal result is the value returned by the function or template in the stylesheet that is nominated as the entry point, as described in [2.3 Initiating a Transformation](#).
2. [**DEFINITION:** Zero or more **secondary results**: each secondary result can be any sequence of items (as defined in [\[XDM 3.0\]](#)).] A secondary result is the value returned by evaluating the body of an [`xsl:result-document`](#) instruction.
3. Zero or more messages. Messages are generated by the [`xsl:message`](#) and [`xsl:assert`](#) instructions, and are described in [23.1 Messages](#) and [23.2 Assertions](#).
4. Static or dynamic errors: see [2.14 Error Handling](#).

The [**principal result**](#) and the [**secondary results**](#) may be post-processed as described in [2.3.6 Post-processing the Raw Result](#).

[**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 output of a transformation: specifically, a tree built by post-processing the items in the [principal result](#) or in a [secondary result](#). Once created, the contents of a final result tree are not accessible within the stylesheet itself.] Any final result tree **MAY** be serialized as described in [26 Serialization](#).

[**DEFINITION:** The term **source tree** means any tree provided as input to the transformation. This includes the document containing the [global context item](#) if any, documents containing nodes present in the [initial match selection](#), documents containing nodes supplied as the values of [stylesheet parameters](#), documents obtained from the results of functions such as [document](#), [doc](#)^{FO30}, and [collection](#)^{FO30}, documents read using the [xsl:source-document](#) instruction, 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.

The use of the term “tree” in phrases such as **source tree**, **result tree**, and **temporary tree** is not confined to documents that the processor materializes in memory in their entirety. The processor **MAY**, and in some cases **MUST**, use streaming techniques to limit the amount of memory used to hold source and result documents. When streaming is used, the nodes of the tree may never all be in memory at the same time, but at an abstract level the information is still modeled as a tree of nodes, and the document is therefore still described as a tree. Unless otherwise stated, the term “tree” refers to a tree rooted at a parentless node: that is, the term does not include subtrees of larger trees. Every node therefore belongs to exactly one tree.

In this specification the phrases **MUST**, **MUST NOT**, **SHOULD**, **SHOULD NOT**, **MAY**, **REQUIRED**, and **RECOMMENDED**, when used in normative text and rendered in capitals, 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 [27 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 3.0](#)] or the XDM specification [[XDM 3.0](#)]. Particular attention is drawn to the following:

- [DEFINITION: The term **atomization** is defined in [Section 2.4.2 Atomization](#)^{XP30}. It is a process that takes as input a sequence of items, and returns a sequence of atomic values, in which the nodes are replaced by their typed values as defined in [[XDM 3.0](#)]. If the [XPath 3.1 Feature](#) is implemented, then arrays (see [27.7.1 Arrays](#)) are atomized by atomizing their members, recursively.] For some items (for example, elements with element-only content, function items, and [maps](#)), atomization generates a [dynamic error](#).
- [DEFINITION: The term **typed value** is defined in [Section 5.15 typed-value Accessor](#)^{DM30}. Every node, other than an element whose type annotation identifies it as having 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](#)^{DM30}. 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](#)^{XP30}. 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 3.0 and XPath 1.0.]

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 the 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.

The types used, and their meanings, are as follows:

boolean

One of the strings "yes", "true", or "1" to indicate the value `true`, or one of the strings "no", "false", or "0" to indicate the value `false`. Note: the values are synonyms; where this specification uses a phrase such as "If `required='yes'` is specified ..." this is to be interpreted as meaning "If the attribute named `required` is present, and has the value yes, true, or 1 (after stripping leading and trailing whitespace) ...".

string

Any string

expression

An XPath [expression](#)

pattern

A [pattern](#) as described in [5.5 Patterns](#).

item-type

An [ItemType](#)^{XP30} as defined in the XPath 3.0 specification (or in XPath 3.1 if the processor implements the [XPath 3.1 Feature](#))

sequence-type

A [SequenceType](#)^{XP30} as defined in the XPath 3.0 specification (or in XPath 3.1 if the processor implements the [XPath 3.1 Feature](#))

uri; uris

A URI, for example a namespace URI or a collation URI; a whitespace-separated list of URIs

qname

A [lexical QName](#) as defined in [5.1.1 Qualified Names](#)

eqname; eqnames

An [EQName](#) as defined in [5.1.1 Qualified Names](#); a whitespace-separated list of EQNames

token; tokens

A string containing no significant whitespace; a whitespace-separated list of such strings

nmtoken; nmtokens

A string conforming to the XML schema rules for the type `xs:NMTOKEN`; a whitespace-separated list of such strings.

char

A string comprising a single Unicode character

integer

An integer, that is a string in the lexical space of the schema type `xs:integer`

decimal

A decimal value, that is a string in the lexical space of the schema type `xs:decimal`

ncname

An unprefixed name: a string in the value space of the schema type `xs:NCName`

prefix

An `xs:NCName` representing a namespace prefix, which must be in scope for the element on which it appears

id

An `xs:NCName` used as a unique identifier for an element in the containing XML document

Except where the set of allowed values of an attribute is specified using the italicized name *string* or *char*, 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.

XPath comments (delimited by (: . . . :)) are permitted anywhere that inter-token whitespace is permitted in attributes whose type is given as *expression*, *pattern*, *item-type*, or *sequence-type*, and are not permitted in attributes of other types (other than within expressions enclosed by curly braces within an [attribute value template](#)).

- 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 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? = boolean
  validation? = { "strict" | "lax" } >
  <!-- 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 the following attributes:

1. A mandatory `select` attribute, whose value is an XPath [expression](#)
2. An optional `debug` attribute, whose value **MUST** be `yes`, `true`, or `1` to indicate `true`, or `no`, `false`, or `0` to indicate `false`.
3. An optional `validation` attribute, whose value must be `strict` or `lax`; the curly brackets indicate that the value can be defined as an [attribute value template](#), allowing a value such as `validation="{$val}"`, where the [variable](#) `val` is evaluated to yield `"strict"` or `"lax"` 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] It is a [static error](#) 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.

The rules in the element syntax summary (both for the element structure and for its attributes) apply to the stylesheet content after preprocessing as described in [3.13 Stylesheet Preprocessing](#).

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 [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 [3.10 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.]

Note:

This specification includes a non-normative XML Schema for XSLT [stylesheet modules](#) (see [H Schemas for XSLT 3.0 Stylesheets](#)). The syntax summaries described in this section are normative.

XSLT defines a set of standard functions which are additional to those defined in [\[Functions and Operators 3.0\]](#). A list of these functions appears in [G.2 List of XSLT-defined functions](#). The signatures of these functions are described using the same notation as used in [\[Functions and Operators 3.0\]](#). The names of many of these functions are 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**.

The execution of a stylesheet necessarily involves two activities: static analysis and dynamic evaluation. Static analysis consists of those tasks that can be performed by inspection of the stylesheet alone, including the binding of [static variables](#), the evaluation of [\[xsl:\]use-when](#) expressions (see [3.13.1 Conditional Element Inclusion](#)), and shadow attributes (see [3.13.2 Shadow Attributes](#)) and detection of [static errors](#). Dynamic evaluation consists of tasks which in general cannot be carried out until a source document is available.

Dynamic evaluation is further divided into two activities: **priming** the stylesheet, and **invoking** a selected component.

- Priming the stylesheet provides the dynamic context for evaluation, and supplies all the information needed to establish the values of global variables.
- Invoking a component (such as a template or function) causes evaluation of that template or function to produce a result, which is an arbitrary XDM value.

[**DEFINITION:** The result of invoking the selected component, after any required conversion to the declared result type of the component, is referred to as the **raw result**.]

The [raw result](#) of the invocation is the [immediate result](#) of evaluating the [sequence constructor](#) contained in the target template or function, modified by applying the [function conversion rules](#) to convert the [immediate result](#) to the type declared in the `as` attribute of the [xsl:template](#) or [xsl:function](#) declaration, if present.

This raw result may optionally be post-processed to construct a result tree, to serialize the result, or both, as described in [2.3.6 Post-processing the Raw Result](#).

Implementations MAY allow static analysis and dynamic evaluation to be initiated independently, so that the cost of static analysis can be amortized over multiple transformations using the same stylesheet. Implementations MAY also allow priming of a stylesheet and invocation of components to be initiated independently, in which case a single act of priming the stylesheet may be followed by a series of independent component invocations. Although this specification does not require such a separation, this section distinguishes information that is needed before static analysis can proceed, information that is needed to prime the stylesheet, and information that is needed when invoking components.

The language is designed to allow the static analysis of each [package](#) to be performed independently of other packages, with only basic knowledge of the properties of components made available by used packages. Beyond this, the specification leaves it to implementations to decide how to organize this process. When packages are not used explicitly, the entire stylesheet is treated as a single package.

2.3.1 Information needed for Static Analysis

The following information is needed prior to static analysis of a package:

- The location of the [package manifest](#), or in the absence of a package manifest, the [stylesheet module](#) that is to act as the [principal stylesheet module](#) of the [package](#). The complete [package](#) is assembled by recursively expanding the [xsl:import](#) and [xsl:include](#) declarations in the principal stylesheet module, as described in [3.11.2 Stylesheet Inclusion](#) and [3.11.3 Stylesheet Import](#).
- Information about the packages referenced from this package using [xsl:use-package](#) declarations. The information needed will include the names and signatures of public components exported by the referenced package.
- A set (possibly empty) of values for [static parameters](#) (see [9.5 Global Variables and Parameters](#)). These values are available for use within [static expressions](#) (notably in [\[xsl:\]use-when](#) expressions and shadow attributes) as well as non-static expressions in the [stylesheet](#). As a minimum, values MUST be supplied for any static parameters declared with the attribute `required="yes"`.

Conceptually, the output of the static analysis of a package is an object which might be referred to (without constraining the implementation) as a compiled package. Prior to dynamic evaluation, all the compiled packages needed for execution must be checked for consistency, and component references must be resolved. This process may be referred to, again without constraining the implementation, as linking.

2.3.2 Priming a Stylesheet

The information needed when priming a stylesheet is as follows:

- A set (possibly empty) of values for non-static [stylesheet parameters](#) (see [9.5 Global Variables and Parameters](#)). These values are available for use within [expressions](#) in the [stylesheet](#). As a minimum, values MUST be supplied for any parameters declared with the attribute `required="yes"`.

A supplied value is converted if necessary to the declared type of the stylesheet parameter using the [function conversion rules](#).

Note:

Non-static stylesheet parameters are implicitly `public`, which ensures that all the parameters in the stylesheet for which values can be supplied externally have distinct names. Static parameters, by contrast, are local to a package.

- [DEFINITION: An item that acts as the **global context item** for the transformation. This item acts as the [context item](#) when evaluating the `select` expression or [sequence constructor](#) of a [global variable](#) declaration within the [top-level package](#), as described in [5.3.3.1 Maintaining Position: the Focus](#). The global context item may also be available in a [named template](#) when the stylesheet is invoked as described in [2.3.4 Call-Template Invocation](#)].

Note:

In previous releases of this specification, a single node was typically supplied to represent the source document for the transformation. This node was used as the target node for the implicit call on `xsl:apply-templates` used to start the transformation process (now called the [initial match selection](#)), and the root node of the containing tree was used as the context item for evaluation of global variables (now called the [global context item](#)). This relationship between the [initial match selection](#) and the [global context item](#) is likely to be found for compatibility reasons in a transformation API designed to work with earlier versions of this specification, but it is no longer a necessary relationship; the two values can in principle be completely independent of each other.

Stylesheet authors wanting to write code that can be invoked using legacy APIs should not rely on the caller being able to supply different values for the [initial match selection](#) and the [global context item](#).

The value given to the [global context item](#) (and the values given to [stylesheet parameters](#)) cannot be nodes in a streamed document. This rule ensures that all global variables can freely navigate within the relevant tree, with no constraints imposed by the streamability rules.

The [global context item](#) is potentially used when initializing global variables and parameters. If the initialization of any [global variables](#) or [parameter](#) depends on the context item, a dynamic error can occur if the context item is absent. It is [implementation-defined](#) whether this error occurs during priming of the stylesheet or subsequently when the variable is referenced; and it is [implementation-defined](#) whether the error occurs at all if the variable or parameter is never referenced. The error can be suppressed by use of `xsl:try` and `xsl:catch` within the sequence constructor used to initialize the variable or parameter. It cannot be suppressed by use of `xsl:try` around a reference to the global variable.

In a [library package](#), the [context item](#), [context position](#), and [context size](#) used for evaluation of global variables will be [absent](#), and the evaluation of any expression that references these values will result in a dynamic error. This will also be the case in the [top-level package](#) if no [global context item](#) is supplied.

Note:

If a context item is available within a global variable declaration, then the [context position](#) and [context size](#) will always be 1 (one).

Note:

For maximum reusability of code, it is best to avoid use of the context item when initializing global variables and parameters. Instead, all external information should be supplied using named [stylesheet parameters](#). Especially when these use namespaces to avoid conflicts, there is then no risk of confusion between the information supplied externally to different packages.

When a stylesheet parameter is defined in a library package, it is possible for a using package to supply a value for the parameter by overriding the parameter declaration within an [xsl:override](#) element. If the using package is the [top-level package](#) then the overriding declaration can refer to the [global context item](#).

- 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^{FO30}](#) 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 [\[XDM 3.0\]](#).

Once a stylesheet is primed, the values of global variables remain stable through all component invocations. In addition, priming a stylesheet creates an [execution scope^{FO30}](#) during which the dynamic context and all calls on [deterministic^{FO30}](#) functions remain stable; for example two calls on the [current-dateTime^{FO30}](#) function within an execution scope are defined to return the same result.

Parameters passed to the transformation by the client application when a stylesheet is primed are matched against [stylesheet parameters](#) (see [9.5 Global Variables and Parameters](#)), not against the [template parameters](#) of any template executed during the course of the transformation.

[ERR XTDE0050] It is a [dynamic error](#) if a stylesheet declares a visible [stylesheet parameter](#) that is [explicitly](#) or [implicitly](#) mandatory, and no value for this parameter is supplied when the stylesheet is primed. A stylesheet parameter is visible if it is not masked by another global variable or parameter with the same name and higher [import precedence](#). If the parameter is a [static parameter](#) then the value **MUST** be supplied prior to the static analysis phase.

[2.3.3 Apply-Templates Invocation](#)

[DEFINITION: A stylesheet may be evaluated by supplying a value to be processed, together with an [initial mode](#). The value (which can be any sequence of items) is referred to as the **initial match selection**. The processing then corresponds to the effect of the [xsl:apply-templates](#) instruction.]

The [initial match selection](#) will often be a single document node, traditionally called the source document of the transformation; but in general, it can be any sequence. If the initial match selection is an empty sequence, the result of the transformation will be empty, since no template rules are evaluated.

Processing proceeds by finding the [template rules](#) that match the items in the [initial match selection](#), and evaluating these template rules with a [focus](#) based on the [initial match selection](#). The template rules are evaluated in [final output state](#).

The following information is needed when dynamic evaluation is to start with a [template rule](#):

- The [initial match selection](#). An API that chooses to maintain compatibility with previous versions of this specification SHOULD allow a method of invocation in which a singleton node is provided, which is then used in two ways: the node itself acts as the [initial match selection](#), and the root node of the containing tree acts as the [global context item](#).
- Optionally, an initial [mode](#).

[DEFINITION: The **initial mode** is the [mode](#) used to select [template rules](#) for processing items in the [initial match selection](#) when [apply-templates](#) invocation is used to initiate a transformation.]

In searching for the [template rule](#) that best matches the items in the [initial match selection](#), the processor considers only those rules that apply to the [initial mode](#).

If no [initial mode](#) is supplied explicitly, then the initial mode is that named in the [default-mode](#) attribute of the (explicit or implicit) [xsl:package](#) element of the [top-level package](#) or in the absence of such an attribute, the [unnamed mode](#).

[ERR XTDE0044] It is a [dynamic error](#) if the invocation of the [stylesheet](#) specifies an [initial mode](#) when no [initial match selection](#) is supplied (either explicitly, or defaulted to the [global context item](#)).

A (named or unnamed) [mode M](#) is **eligible as an initial mode** if one of the following conditions applies, where P is the [top-level package](#) of the stylesheet:

1. M is explicitly declared in an [xsl:mode](#) declaration within P , and has [public](#) or [final visibility](#) (either by virtue of its [visibility](#) attribute, or by virtue of an [xsl:expose](#) declaration).
2. M is the unnamed mode.
3. M is named in the [default-mode](#) attribute of the (explicit or implicit) [xsl:package](#) element of P .
4. M is declared in a package used by P , and is given [public](#) or [final visibility](#) in P by means of an [xsl:accept](#) declaration.
5. The effective value of the [declared-modes](#) attribute of the explicit or implicit [xsl:package](#) element of P is no, and M appears as a mode-name in the [mode](#) attribute of a [template rule](#) declared within P .

[ERR XTDE0045] It is a [dynamic error](#) if the invocation of the [stylesheet](#) specifies an [initial mode](#) and the specified mode is not eligible as an initial mode (as defined above).

- Parameters, which will be passed to the template rules used to process items in the input sequence. The parameters consist of two sets of (QName, value) pairs, one set for [tunnel parameters](#) and one for non-tunnel parameters, in which the QName identifies the name of a parameter and the value provides the value of the parameter. Either or both sets of parameters may be empty. The effect is the same as when a template is invoked using [xsl:apply-templates](#) with an [xsl:with-param](#) child specifying [tunnel="yes"](#) or [tunnel="no"](#) as appropriate. If a parameter is supplied that is not declared or used, the value is simply ignored. These parameters are *not* used to set [stylesheet parameters](#).

A supplied value is converted if necessary to the declared type of the template parameter using the [function conversion rules](#).

- Details of how the result of the initial template is to be returned. For details, see [2.3.6 Post-processing the Raw Result](#)

The [raw result](#) of the invocation is the result of processing the supplied input sequence as if by a call on [`xsl:apply-templates`](#) in the specified mode: specifically, each item in the input sequence is processed by selecting and evaluating the best matching template rule, and converting the result (if necessary) to the type declared in the `as` attribute of that template using the [function conversion rules](#); and the results of processing each item are then concatenated into a single sequence, respecting the order of items in the input sequence.

Note:

If the initial mode is [declared-streamable](#), then a streaming processor SHOULD allow some or all of the items in the [initial match selection](#) to be nodes supplied in streamable form, and any nodes that are supplied in this form MUST then be processed using streaming.

Since the [global context item](#) cannot be a streamed node, in cases where the transformation is to proceed by applying streamable templates to a streamed input document, the [global context item](#) must either be absent, or must be something that differs from the [initial match selection](#).

Note:

The design of the API for invoking a transformation should provide some means for users to designate the [unnamed mode](#) as the [initial mode](#) in cases where it is not the default mode.

It is a [dynamic error](#) [see [ERR_XTDE0700](#)] if the [template rule](#) selected for processing any item in the [initial match selection](#) defines a [template parameter](#) that specifies `required="yes"` and no value is supplied for that parameter.

Note:

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 [20.1 fn:document](#)) or [`doc`](#)^{FO30} or [`collection`](#)^{FO30} (see [\[Functions and Operators 3.0\]](#)), or using the [`xsl:source-document`](#) instruction; alternatively, they can be supplied as [stylesheet parameters](#) (see [9.5 Global Variables and Parameters](#)), or returned as the result of an [extension function](#) (see [24.1 Extension Functions](#)).

2.3.4 Call-Template Invocation

[**DEFINITION:** A stylesheet may be evaluated by selecting a named template to be evaluated; this is referred to as the **initial named template**.] The effect is analogous to the effect of executing an [`xsl:call-template`](#) instruction. The following information is needed in this case:

- Optionally, the name of the [initial named template](#) which is to be executed as the entry point to the transformation. If no template name is supplied, the default template name is [`xsl:initial-template`](#). The selected template **MUST** exist within the [stylesheet](#).
- Optionally, a context item for evaluation of this named template, defaulting to the [global context item](#) if it exists. This is constrained by any [`xsl:context-item`](#) element appearing within the selected [`xsl:template`](#) element. The initial named template is evaluated with a [singleton focus](#) based on this context item if it exists, or with an [absent](#) focus otherwise.

- Parameters, which will be passed to the selected template rule. The parameters consist of two sets of (QName, value) pairs, one set for [tunnel parameters](#) and one for non-tunnel parameters, in which the QName identifies the name of a parameter and the value provides the value of the parameter. Either or both sets of parameters may be empty. The effect is the same as when a template is invoked using [`xsl:call-template`](#) with an [`xsl:with-param`](#) child specifying `tunnel="yes"` or `tunnel="no"` as appropriate. If a parameter is supplied that is not declared or used, the value is simply ignored. These parameters are *not* used to set [stylesheet parameters](#).

A supplied value is converted if necessary to the declared type of the template parameter using the [function conversion rules](#).

- Details of how the result of the initial named template is to be returned. For details, see [2.3.6 Post-processing the Raw Result](#)

The [raw result](#) of the invocation is the result of evaluating the [initial named template](#), after conversion of the result to the type declared in the `as` attribute of that template using the [function conversion rules](#), if such conversion is necessary.

The [initial named template](#) is evaluated in [final output state](#).

[ERR XTDE0040] It is a [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](#), whose visibility is `public` or `final`.

It is a [dynamic error](#) [see [ERR XTDE0700](#)] if the [initial named template](#), or any of the template rules invoked to process items in the [initial match selection](#), defines a [template parameter](#) that specifies `required="yes"` and no value is supplied for that parameter.

[2.3.5 Function Call Invocation](#)

[**DEFINITION:** A stylesheet may be evaluated by calling a named [stylesheet function](#), referred to as the **initial function**.] The following additional information is needed in this case:

- The name and arity of a [stylesheet function](#) which is to be executed as the entry point to the transformation.

Note:

In the design of a concrete API, the arity may be inferred from the length of the parameter list.

- A list of values to act as parameters to the [initial function](#). The number of values in the list must be the same as the arity of the function.
A supplied value is converted if necessary to the declared type of the function parameter using the [function conversion rules](#).
- Details of how the result of the initial function is to be returned. For details, see [2.3.6 Post-processing the Raw Result](#)

The [raw result](#) of the invocation is the result of evaluating the [initial function](#), after conversion of the result to the type declared in the `as` attribute of that function using the [function conversion rules](#), if such conversion is necessary.

Note:

The [initial function](#) (like all stylesheet functions) is evaluated with an [absent focus](#).

If the [initial function](#) is [declared-streamable](#), a streaming processor SHOULD allow the value of the first argument to be supplied in streamable form, and if it is supplied in this form, then it MUST be processed using streaming.

[ERR XTDE0041] It is a [dynamic error](#) if the invocation of the [stylesheet](#) specifies a function name and arity that does not match the [expanded QName](#) and arity of a named [stylesheet function](#) defined in the [stylesheet](#), whose visibility is [public](#) or [final](#).

When a transformation is invoked by calling an [initial function](#), the entire transformation executes in [temporary output state](#), which means that calls on [xsl:result-document](#) are not permitted.

2.3.6 Post-processing the Raw Result

There are three ways the result of a transformation may be delivered. (This applies both to the principal result, described here, and also to secondary results, generated using [xsl:result-document](#).)

1. The [raw result](#) (a sequence of values) may be returned directly to the calling application.
2. A result tree may be constructed from the [raw result](#). By default, a result tree is constructed if the [build-tree](#) attribute of the unnamed [output definition](#) has the effective value [yes](#). An API for invoking transformations MAY allow this setting to be overridden by the calling application. If result tree construction is requested, it is performed as described in [2.3.6.1 Result Tree Construction](#).
3. Alternatively, the [raw result](#) may be serialized as described in [2.3.6.2 Serializing the Result](#). The decision whether or not to serialize the result is determined by the rules of transformation API provided by the [processor](#), and is not influenced by anything in the stylesheet.

Note:

This specification does not constrain the design of application programming interfaces or the choice of defaults. In previous versions of this specification, result tree construction was a mandatory process, while serialization was optional. When invoking stylesheet functions directly, however, result tree construction and serialization may be inappropriate as defaults. These considerations may affect the design of APIs.

In previous versions of XSLT, results were delivered either in serialized form (as a character or byte stream), or as a tree. In the latter case processors typically would use either their own tree representation, or a standardized tree representation such as the W3C Document Object Model (DOM) (see [\[DOM Level 2\]](#)), adapted to the data structures offered by the programming language in which the API is defined. To deliver a raw result, processors need to define a representation not only of XDM nodes but also of sequences, atomic values, maps and even functions. As with the return of a simple tree, this may involve a trade-off between strict fidelity to the XDM data model and usability in the particular programming language environment. It is *not* a requirement that an API should return results in a way that exposes every property of the XDM data model; for example there may be APIs that do not expose the precise type annotation of a returned node or atomic value, or that fail to expose the base URI or document URI of a node, or that provide no way of determining whether two nodes in the result sequence are the same node in the sense of the XPath `is` operator. The way in which maps and functions (and where XPath 3.1 is supported, arrays) are returned requires careful design choices. It is RECOMMENDED that an API should be capable of returning any XDM value without error, and that there should be minimal loss of information if the raw results output by one transformation are subsequently used as input to another transformation.

2.3.6.1 [Result Tree Construction](#)

If a result tree is to be constructed from the [raw result](#), then this is done by applying the rules for the process of [sequence normalization](#)^{SER30} as defined in [\[XSLT and XQuery Serialization\]](#). This process takes as input the serialization parameters defined in the unnamed [output definition](#) of the [top-level package](#); though the only parameter that is actually used by this process is `item-separator`. In particular, sequence normalization is carried out regardless of any `method` attribute in the unnamed [output definition](#).

The sequence normalization process either returns a document node, or raises a serialization error. The content of the document node is not necessarily well-formed (the document node may have any number of element or text nodes among its children).

Note:

More specifically, the process raises a serialization error if any item in the [raw result](#) is an attribute node, a namespace node, or a function (including a map, but not an array: arrays are flattened).

The tree that is constructed is referred to as a [final result tree](#).

If the [raw result](#) is an empty sequence, the [final result tree](#) will consist of a document node with no children.

The base URI of the document node is set to the [base output URI](#).

Note:

The `item-separator` property has no effect if the raw result of the transformation is a sequence of length zero or one, which in practice will often be the case, especially in a traditional scenario such as transformation of an XML document to HTML.

If there is no `item-separator`, then a single space is inserted between adjacent atomic values; for example if the raw result is the sequence 1 to 5, then sequence normalization produces a tree comprising a document node with a single child, the child being a text node with the string value 1 2 3 4 5.

If there is an `item-separator`, then it is used not only between adjacent atomic values, but between any pair of items in the raw result. For example if the raw result is a sequence of two element nodes A and B, and the `item-separator` is a comma, then the result of sequence normalization will be a document node with three children: a copy of A, a text node whose string value is a single comma, and a copy of B.

2.3.6.2 *Serializing the Result*

See [2.7 Parsing and Serialization](#).

The `raw result` may optionally be serialized as described in [26 Serialization](#). The serialization is controlled by the serialization parameters defined in the unnamed [output definition](#) of the [top-level package](#).

Note:

The first phase of serialization, called [sequence normalization^{SER30}](#), takes place for some output methods but not others. For example, if the `json` output method (defined in [\[XSLT and XQuery Serialization 3.1\]](#)) is selected, then the process of constructing a tree is bypassed.

The effect of serialization is to generate a sequence of octets, representing the serialized result in some character encoding. The processor's API may define mechanisms enabling this sequence of octets to be written to persistent storage at some location. The default location is the location identified by the [base output URI](#).

In previous versions of this specification it was stated that when the `raw result` of the initial template or function is an empty sequence, a result tree should be produced if and only if the transformation generates no secondary results (that is, if it does not invoke [xsl:result-document](#)). This provision is most likely to have a noticeable effect if the transformation produces serialized results, and these results are written to persistent storage: the effect is then that a transformation producing an empty principal result will overwrite any existing content at the base output URI location if and only if the transformation produces no other output. Processor APIs offering backwards compatibility with earlier versions of XSLT must respect this behavior, but there is no requirement for new processor APIs to do so.

[**DEFINITION:** The **base output URI** is a URI to be used as the base URI when resolving a relative URI reference 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](#). Each invocation of the stylesheet may supply a different base output URI. It is acceptable for the base output URI to be [absent](#), provided no constructs (such as [xsl:result-document](#)) are evaluated that depend on the value of the base output URI.

Note:

It will often be convenient for the base output URI to be the same as the location to which the principal result document is serialized, but this relationship is not a necessary one.

2.4 Instructions

The main executable components of a stylesheet are templates and functions. The body of a template or function is a [sequence constructor](#), which is a sequence of elements and text nodes that can be evaluated to produce a result.

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 [24.2 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](#);
- instructions that copy nodes: [xsl:copy](#), [xsl:copy-of](#);
- 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:try](#), [xsl:for-each](#), [xsl:for-each-group](#), [xsl:fork](#), [xsl:iterate](#) and its subordinate instructions [xsl:next-iteration](#) and [xsl:break](#);
- instructions that generate output conditionally if elements are or are not empty: [xsl:on-empty](#), [xsl:on-non-empty](#), [xsl:where-populated](#);
- instructions that invoke templates: [xsl:apply-templates](#), [xsl:apply-imports](#), [xsl:call-template](#), [xsl:next-match](#);
- Instructions that declare variables: [xsl:variable](#);
- Instructions to assist debugging: [xsl:message](#), [xsl:assert](#);
- other specialized instructions: [xsl:number](#), [xsl:analyze-string](#), [xsl:fork](#), [xsl:result-document](#), [xsl:source-document](#), [xsl:perform-sort](#), [xsl:merge](#).

2.5 Rule-Based Processing

The classic method of executing an XSLT transformation is to apply template rules to the root node of an input document (see [2.3.3 Apply-Templates Invocation](#)). The operation of applying templates to a node searches the stylesheet for the best matching template rule for that node. This template rule is then evaluated. A common coding pattern, especially when XSLT is used to convert XML documents into display formats such as HTML, is to have one template rule for each kind of element in the source document, and for that template rule to generate some

appropriate markup elements, and to apply templates recursively to its own children. The effect is to perform a recursive traversal of the source tree, in which each node is processed using the best-fit template rule for that node. The final result of the transformation is then the tree produced by this recursive process. This result can then be optionally serialized (see [2.3.6 Post-processing the Raw Result](#)).

Example: Example of Rule-Based Processing

This example uses rule-based processing to convert a simple XML input document into an HTML output document.

The input document takes the form:

```
<PERSONAE PLAY="OTHELLO">
    <TITLE>Dramatis Personae</TITLE>
    <PERSONA>DUKE OF VENICE</PERSONA>
    <PERSONA>BRABANTIO, a senator.</PERSONA>
    <PERSONA>Other Senators.</PERSONA>
    <PERSONA>GRATIANO, brother to Brabantio.</PERSONA>
    <PERSONA>LODOVICO, kinsman to Brabantio.</PERSONA>
    <PERSONA>OTHELLO, a noble Moor in the service of the Venetian state.
    </PERSONA>
    <PERSONA>CASSIO, his lieutenant.</PERSONA>
    <PERSONA>IAGO, his ancient.</PERSONA>
    <PERSONA>RODERIGO, a Venetian gentleman.</PERSONA>
    <PERSONA>MONTANO, Othello's predecessor in the government of Cyprus.
    </PERSONA>
    <PERSONA>Clown, servant to Othello. </PERSONA>
    <PERSONA>DESDEMONA, daughter to Brabantio and wife to Othello.</PERSONA>
    <PERSONA>EMILIA, wife to Iago.</PERSONA>
    <PERSONA>BIANCA, mistress to Cassio.</PERSONA>
    <PERSONA>Sailor, Messenger, Herald, Officers,
        Gentlemen, Musicians, and Attendants.</PERSONA>
</PERSONAE>
```

The stylesheet to render this as HTML can be written as a set of template rules:

```

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="3.0"
    expand-text="yes">

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

    <xsl:template match="PERSONAE">
        <html>
            <head>
                <title>The Cast of {@PLAY}</title>
            </head>
            <body>
                <xsl:apply-templates/>
            </body>
        </html>
    </xsl:template>

    <xsl:template match="TITLE">
        <h1>{.}</h1>
    </xsl:template>

    <xsl:template match="PERSONA[count(tokenize(., ',')) = 2]">
        <p><b>{substring-before(., ',')}:{substring-after(., ',')}</p>
    </xsl:template>

    <xsl:template match="PERSONA">
        <p><b>{.}</b></p>
    </xsl:template>

</xsl:stylesheet>

```

There are four template rules here:

- The first rule matches the outermost element, named PERSONAE (it could equally have used `match="/"` to match the document node). The effect of this rule is to create the skeleton of the output HTML page. Technically, the body of the template is a sequence constructor comprising a single [literal result element](#) (the `html` element); this in turn contains a sequence constructor comprising two literal result elements (the `head` and `body` elements). The `head` element is populated with a literal `title` element whose content is computed as a mixture of fixed and variable text using a [text value template](#). The `body` element is populated by evaluating an [`xsl:apply-templates`](#) instruction.
The effect of the [`xsl:apply-templates`](#) instruction is to process the children of the PERSONAE element in the source tree: that is, the TITLE and PERSONA elements. (It would also process any whitespace text node children, but these have been stripped by virtue of the [`xsl:strip-space`](#) declaration.) Each of these child elements is processed by the best matching template rule for that element, which will be one of the other three rules in the stylesheet.
- The template rule for the TITLE element outputs an `h1` element to the HTML result document, and populates this with the value of `"."`, the context item. That is, it copies the text content of the TITLE element to the output `h1` element.
- The last two rules match PERSONA element. The first rule matches PERSONA elements whose text content contains exactly one comma; the second rule matches all PERSONA elements, but it has lower priority than

the first rule, so in practice it only applies to PERSONA elements that contain no comma or multiple commas.

For both rules the body of the rule is a sequence constructor containing a single literal result element, the p element. These literal result elements contain further sequence constructors comprising literal result elements and text nodes. In each of these examples the text nodes are in the form of a [text value template](#): in general this is a combination of fixed text together with XPath expressions enclosed in curly braces, which are evaluated to form the content of the containing literal result element.

[**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 selected items (often but not necessarily 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](#).

[2.6 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 3.0 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 [current-group](#)) 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 currently being processed (it may be a node, an atomic value, or a function item). 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 [5.3 The Static and Dynamic Context](#).

2.7 Parsing and Serialization

An XSLT [stylesheet](#) describes a process that constructs a set of results from a set of inputs. The inputs are the data provided at stylesheet invocation, as described in [2.3 Initiating a Transformation](#). The results include the [principal result](#) (an arbitrary sequence), which is the result of the initial component invocation, together with any [secondary results](#) produced using [`xsl:result-document`](#) instructions.

The [stylesheet](#) does not describe how a [source tree](#) is constructed. Some possible ways of constructing source trees are described in [\[XDM 3.0\]](#). 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 [\[XDM 3.0\]](#), 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 [26 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.

2.8 Packages and Modules

In previous versions of the XSLT language, it has been possible to structure a stylesheet as a collection of modules, using the [`xsl:include`](#) and [`xsl:import`](#) declarations to express the dependency of one module on others.

In XSLT 3.0 an additional layer of modularization of stylesheet code is enabled through the introduction of [packages](#). A package is a collection of stylesheet modules with a controlled interface to the packages that use it: for example, it defines which functions and templates defined in the package are visible to callers, which are purely internal, and which are not only public but capable of being overridden by other functions and templates supplied by the using package.

Packages are introduced with several motivations, which broadly divide into two categories:

1. Software engineering benefits: greater re-use of code, greater robustness through ease of testing, controlled evolution of code in response to new requirements, ability to deliver code that users cannot see or modify.
2. Efficiency benefits: the ability to avoid compiling libraries repeatedly when they are used in multiple stylesheets, and to avoid holding multiple copies of the same library in memory simultaneously.

Packages are designed to allow separate compilation: that is, a package can be compiled independently of the packages that use it. This specification does not define a process model for compilation, or expand on what it means to compile different packages independently. Nor does it mandate that implementations offer any feature along these lines. It merely defines language features that are designed to make separate compilation of packages possible.

To achieve this, packages (unlike modules):

- Must not contain unresolved references to functions, templates, or variables declared in other packages;
- Have strict rules governing the ability to override declarations in a [library package](#) with declarations in a package that uses the library;
- Constrain the visibility of component names and of context declarations such as the declarations of keys and decimal formats;
- Can declare a mode (a collection of template rules) as final, which disallows the addition of new overriding template rules in a using package;
- Require explicit disambiguation where naming conflicts arise, for example when a package uses two other packages that both export like-named components;
- Allow multiple specializations of library components to coexist in the same application.

A package is defined in XSLT by means of an XML document whose outermost element is an [xsl:package](#) element. This is referred to as the [package manifest](#). The [xsl:package](#) element has optional child elements [xsl:use-package](#) and [xsl:expose](#) describing properties of the package. The package manifest may refer to an external top-level stylesheet module using an [xsl:include](#) or [xsl:import](#) declaration, or it may contain the body of a stylesheet module inline (the two approaches can also be mixed).

Although this specification defines packages as constructs written using a defined XSLT syntax, implementations MAY provide mechanisms that allow packages to be written using other languages (for example, XQuery).

When no packages are explicitly defined, the entire stylesheet is treated as a single package; the effect is as if the [xsl:stylesheet](#) or [xsl:transform](#) element of the [principal stylesheet module](#) were replaced by an [xsl:package](#) element with no other information in the package manifest.

[2.9 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 [24.1 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 [24.2 Extension Instructions](#). A stylesheet that uses extension instructions in a

particular namespace 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 [24 Extensibility and Fallback](#).

The XSLT language can also be extended by the use of [extension attributes](#) (see [3.2 Extension Attributes](#)), and by means of [user-defined data elements](#) (see [3.7.3 User-defined Data Elements](#)).

[2.10 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 a [SequenceType](#).

[**DEFINITION:** A **SequenceType** constrains the type and number of items in a sequence. The term is used both to denote the concept, and to refer to the syntactic form in which sequence types are expressed in the XPath grammar: specifically [SequenceType^{XP30}](#) in [\[XPath 3.0\]](#), or [SequenceType^{XP31}](#) in [\[XPath 3.1\]](#), depending on whether or not the [XPath 3.1 Feature](#) is implemented.]

[**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 [package](#) are referred to as the **in-scope schema components**.]

The set of in-scope schema components may vary between one package and another, but as explained in [3.15 Importing Schema Components](#), the schema components used in different packages must be consistent with each other.

The conformance rules for XSLT 3.0, defined in [27 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`^{FO30} function) and other constructs may produce different results depending on the datatype (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 [3.14 Built-in Types](#).

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.

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:mode typed="lax"/>
<xsl:global-context-item use="required"
    as="document-node(schema-element(my:invoice))"/>
```

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

The setting `typed="lax"` further ensures that in any match pattern for a template rule in this mode, an element name that corresponds to the name of an element declaration in the schema is taken as referring to elements validated against that declaration: for example, `match="employee"` will only match a validated `employee` element. Selecting this option enables the XSLT processor to do more compile-time type-checking against the schema, for example it allows the processor to produce warning or error messages when path expressions contain misspelt element names, or confuse an element with an attribute.

It is also true that importing a schema does not of itself say anything about the structure of the result tree. It is possible to request validation of a result tree against the schema by using the xsl:result-document instruction, for example:

Example: Requesting Validation of the Result Document

```
<xsl:template match="/">
  <xsl:result-document validation="strict">
    <xhtml:html>
      <xsl:apply-templates/>
    </xhtml:html>
  </xsl:result-document>
</xsl:template>
```

This example will cause the transformation to fail with an error message unless the document element of the result document is valid against the top-level element declaration `xhtml:html`.

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 [\[XDM 3.0\]](#). The implementation may choose one of several strategies for dealing with this situation:

1. The processor may signal a [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 [\[XDM 3.0\]](#), 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 [2.14 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 [result tree](#). This can be done in a number of ways.

- It is possible to request explicit validation of a complete document, that is, a [result tree](#) rooted at a document node. 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 [25.4 Validation](#).

[2.11 Streaming](#)

[**DEFINITION:** The term **streaming** refers to a manner of processing in which XML documents (such as source and result documents) are not represented by a complete tree of nodes occupying memory proportional to document size, but instead are processed “on the fly” as a sequence of events, similar in concept to the stream of events notified by an XML parser to represent markup in lexical XML.]

[**DEFINITION:** A **streamed document** is a [`source tree`](#) that is processed using streaming, that is, without constructing a complete tree of nodes in memory.]

[**DEFINITION:** A **streamed node** is a node in a [`streamed document`](#).]

Many processors implementing earlier versions of this specification have adopted an architecture that allows streaming of the [`result tree`](#) directly to a serializer, without first materializing the complete result tree in memory. Streaming of the [`source tree`](#), however, has proved to be more difficult without subsetting the language. This has created a situation where documents exceeding the capacity of virtual memory could not be transformed. XSLT 3.0 therefore introduces facilities allowing stylesheets to be written in a way that makes streaming of source documents possible, without excessive reliance on processor-specific optimization techniques.

Streaming achieves two important objectives: it allows large documents to be transformed without requiring correspondingly large amounts of memory; and it allows the processor to start producing output before it has finished receiving its input, thus reducing latency.

This specification does not attempt to legislate precisely which implementation techniques fall under the definition of streaming, and which do not. A number of techniques are available that reduce memory requirements, while still requiring a degree of buffering, or allocation of memory to partial results. A stylesheet that requests streaming of a source document is indicating that the processor should avoid assuming that the entire source document will fit in memory; in return, the stylesheet must be written in a way that makes streaming possible. This specification does not attempt to describe the algorithms that the processor should actually use, or to impose quantitative constraints on the resources that these algorithms should consume.

Nothing in this specification, nor in its predecessors [\[XSLT 1.0\]](#) and [\[XSLT 2.0\]](#), prevents a processor using streaming whenever it sees an opportunity to do so. However, experience has shown that in order to achieve streaming, it is often necessary to write stylesheet code in such a way as to make this possible. Therefore, XSLT 3.0

provides explicit constructs allowing the stylesheet author to request streaming, and defines explicit static constraints on the structure of the code which are designed to make streaming possible.

A processor that claims conformance with the streaming option offers a guarantee that when streaming is requested for a source document, and when the stylesheet conforms to the rules that make the processing guaranteed-streamable, then an algorithm will be adopted in which memory consumption is either completely independent of document size, or increases only very slowly as document size increases, allowing documents to be processed that are orders-of-magnitude larger than the physical memory available. A processor that does not claim conformance with the streaming option must still process a stylesheet and deliver the correct results, but is not required to use streaming algorithms, and may therefore fail with out-of-memory errors when presented with large source documents.

Apart from the fact that there are constructs to request streaming, and rules that must be followed to guarantee that streaming is possible, the language has been designed so there are as few differences as possible between streaming and non-streaming evaluation. The semantics of the language continue to be expressed in terms of the XDM data model, which is substantively unchanged; but readers must take care to observe that when terms like “node” and “axis” are used, the concepts are completely abstract and may have no direct representation in the run-time execution environment.

Streamed processing of a document can be initiated in one of three ways:

- The initial mode can be declared as a streamable mode. In this case the initial match selection will generally be a document node (or sequence of document nodes), supplied by the calling application in a form that allows streaming (that is, in some form other than a tree in memory; for example, as a reference to a push or pull XML parser primed to deliver a stream of events). The type of these nodes can be constrained by using the attribute `on-no-match="fail"` on the initial mode, and using this mode only for processing the top-level nodes.
- Streamed processing of any document can be initiated using the `xsl:source-document` instruction. This has an attribute `href` whose value is the URI of a document to be processed, and an attribute `streamable` that indicates whether it is to be processed using streaming; the actual processing to be applied is defined by the instructions written as children of the `xsl:source-document` instruction.
- Streamed merging of a set of input documents can be initiated using the `xsl:merge` instruction.

The rules for streamability, which are defined in detail in [19 Streamability](#), impose two main constraints:

- The only nodes reachable from the node that is currently being processed are its attributes and namespaces, its ancestors and their attributes and namespaces, and its descendants and their attributes and namespaces. The siblings of the node, and the siblings of its ancestors, are not reachable in the tree, and any attempt to use their values is a static error.
- When processing a given node in the tree, each descendant node can only be visited once. Essentially this allows two styles of processing: either visit each of the children once, and then process that child with the same restrictions applied; or process all the descendants in a single pass, in which case it is not possible while processing a descendant to make any further downward selection.

The second restriction, that only one visit to the children is allowed, means that XSLT code that was not designed with streaming in mind will often need to be rewritten to make it streamable. In many cases it is possible to do this using a technique sometimes called *windowing* or *burst-mode streaming* (note this is not quite the same meaning as *windowing* in XQuery 3.0). Many XML documents consist of a large number of elements, each of manageable size, representing transactions or business objects where each such element can be processed independently: in such

cases, an effective design pattern is to write a streaming transformation that takes a snapshot of each element in turn, processing the snapshot using the full power of the XSLT language. Each snapshot is a tree built in memory and is therefore fully navigable. For details see the [snapshot](#) and [copy-of](#) functions.

The new facility of *accumulators* allows applications complete control over how much information is retained (and by implication, how much memory is required) in the course of a pass over a [streamed document](#). An accumulator computes a value for every node in a streamed document: or more accurately, two values, one for the first visit to a node (before visiting its descendants), and a second value for the second visit to the node (after visiting the descendants). The computation is structured in such a way that the value for a given node can depend only on the value for the previous node in document order together with the data available when positioned at the current node (for example, the attribute values). Based on the well-established fold operation of functional programming languages, accumulators provide the convenience and economy of mutable variables while remaining within the constraints of a purely declarative processing model.

When streaming is initiated, for example using the [xsl:source-document](#) instruction, it is necessary to declare which accumulators are applicable to the streamed document.

Streaming applications often fall into one of the following categories:

- Aggregation applications, where a single aggregation operation (perhaps [count](#)^{FO30}, [sum](#)^{FO30}, [exists](#)^{FO30}, or [distinct-values](#)^{FO30}) is applied to a set of elements selected from the streamed source document by means of a path expression.
- Record-at-a-time applications, where the source document consists of a long sequence of elements with similar structure (“records”), and each “record” is processed using the same logic, independently of any other “records”. This kind of processing is facilitated using the [snapshot](#) and [copy-of](#) function mentioned earlier.
- Grouping applications, where the output follows the structure of the input, except that an extra layer of hierarchy is added. For example, the input might be a flat series of banking transactions in date/time order, and the output might contain the same transactions grouped by date.
- Accumulator applications, which are the same as record-at-a-time applications, except that the processing of one “record” might depend on data encountered earlier in the document. A classic example is processing a sequence of banking transactions in which the input transaction contains a debit or credit amount, and the output adds a running total (the account balance). The [xsl:iterate](#) instruction has been introduced to facilitate this style of processing.
- Isomorphic transformations, in which there is an ordered (often largely one-to-one) relationship between the nodes of the source tree and the nodes of the result tree: for example, transformations that involve only the renaming or selective deletion of nodes, or scalar manipulations of the values held in the leaf nodes. Such transformations are most conveniently expressed using recursive application of template rules. This is possible with a streamed input document only if all the template rules adhere to the constraints required for streamability. To enforce these rules, while still allowing unrestricted processing of other documents within the same transformation, all streaming evaluation must be carried out using a specific [mode](#), which is declared to be a streaming mode by means of an [xsl:mode](#) declaration in the stylesheet.

There are important classes of application in which streaming is possible only if multiple streams can be processed in parallel. This specification therefore provides facilities:

1. allowing multiple sorted input sequences to be merged into one sorted output sequence (the [xsl:merge](#) instruction)

2. allowing multiple output sequences to be generated during a single pass of an input sequence (the [xsl:fork](#) instruction).

These facilities have been designed in such a way that they can readily be implemented using streaming, that is, without materializing the input or output sequences in memory.

[2.12 Streamed Validation](#)

Streaming can be combined with schema-aware processing: that is, the streamed input to a transformation can be subjected to on-the-fly validation, a process which typically accepts an input stream from the XML parser and delivers an output stream (of type-annotated nodes) to the transformation processor. The XSD specification is designed so that validation is, with one or two exceptions, a streamable process. The exceptions include:

- There may be a need to allocate memory to hold keys, in order to enforce uniqueness and referential integrity constraints (`xs:unique`, `xs:key`, `xs:keyref`).
- In XSD 1.1, assertions can be defined by means of XPath expressions. These are not constrained to be streamable; in the general case, any subtree of the document that is validated using an assertion may need to be buffered in memory while the assertion is processed.

Applications that need to run in finite memory may therefore need to avoid these XSD features, or to use them with care.

XSD is designed so that the intended type of an element (the “governing type”) can be determined as soon as the start tag of the element is encountered: the process of validation checks whether the content of the element actually conforms to this type, and by the time the end tag is encountered, the process will have established either that the element is valid against the governing type, or that it is invalid.

By default, dynamic errors occurring during streamed processing are fatal: they typically cause the transformation to fail immediately. XSLT 3.0 introduces the ability to catch dynamic errors and recover from them. Schema invalidity, however, is treated as a dynamic error of the instruction that processes the entire input stream, so after a validation failure, no further processing of that input stream is possible.

In consequence, a streamed validator that is running in tandem with a streamed transformation can present the transformer with element nodes that carry a provisional type annotation representing the type that the element will have if it turns out to be valid. As soon as a node is encountered that violates this assumption, the validator should stop the flow of data to the transformer, so that the transformer never sees invalid data. This allows the stylesheet code to be compiled with the assumption of type-safety: at run-time, all nodes seen by the transformation will conform to their XSLT-declared types (for example, a type declared implicitly using `match="schema-element(invoice)"` on an [xsl:template](#) element).

A streamed transformation that only accesses part of the input document (for example, a header at the start of a document) is not required to continue reading once the data it needs has been read. This means that XML well-formedness or validity errors occurring in the unread part of the input stream may go undetected.

[2.13 Streaming of non-XML data](#)

The facilities in this specification designed to enable large data sets to be processed in a streaming manner are oriented almost entirely to XML data. This does not mean that there is never a requirement to stream non-XML

data, or that the Working Group has ignored this requirement; rather, the Working Group has concluded that for the most part, streaming of non-XML data can be achieved by implementations without the need for specific language features in XSLT.

To make streamed processing of unparsed text files easier, the function [unparsed-text-lines^{FO30}](#) has been introduced. This is not only more convenient for stylesheet authors than reading the entire input using the [unparsed-text^{FO30}](#) function and then tokenizing the result, it is also easier for implementations to optimize, allowing each line of text to be discarded from memory after it has been processed.

For all functions that access external data, including [document](#), [doc^{FO30}](#), [collection^{FO30}](#), [unparsed-text^{FO30}](#), [unparsed-text-lines^{FO30}](#), and (in XPath 3.1) [json-doc^{FO31}](#), the requirements on determinism can now be relaxed using [implementation-defined](#) configuration options. This is significant because it means that when a transformation reads the same external resource more than once, it becomes legitimate for the contents of the resource to be different on different invocations, and this eliminates the need for the processor to cache the contents of the resource in memory.

In the XDM data model, every value is a sequence, and (as with most functional programming languages), processing of sequences of items is pervasive throughout the XSLT and XPath languages and their function library. Good performance of a functional programming language often depends on sequence-based operations being pipelined, and being evaluated in a lazy fashion (that is, many operations process items in a sequence one at a time, in order; and many operations can deliver a result without processing the entire sequence). The semantics of XSLT and XPath permit pipelined and lazy evaluation (for example, the error handling semantics are carefully written to ensure this), but they do not require it: the details are left to implementations. Pipelined processing of a sequence is not the same thing as streamed processing of a tree, and where the XSLT specification talks of operations being “guaranteed streamable”, this is always referring to processing of trees, not of sequences.

The facilities for streaming of XML trees include operations such as [copy-of](#) and [snapshot](#) which are able to take a sequence of streamed nodes as input, and produce a sequence of in-memory (unstreamed) nodes as output. It is also possible to generate a sequence of strings or other atomic values through the process of [atomization](#). The actual memory usage of a streamed XSLT application may depend significantly on whether the processing of the resulting sequence of in-memory nodes or atomic values is pipelined or not. The specification, however, has nothing to say on this matter: it is considered an area where implementers can exercise their discretion and ingenuity.

Streaming of JSON input receives little attention in this specification. One can envisage an implementation of the [json-to-xml](#) function in which the XML delivered by the function consists of streamed nodes; but the Working Group has not researched the feasibility of such an implementation in any detail.

2.14 Error Handling

[**DEFINITION:** An error that can be 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**.]

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.

A processor **MUST** provide a mode of operation that takes a (possibly erroneous) stylesheet [package](#) as input and enables the user to determine whether or not that package contains any [static errors](#).

Note:

The manner in which static errors are reported, and the behavior when there are multiple static errors, are left as design choices for the implementer. It is **RECOMMENDED** that the error codes defined in this specification should be made available in any diagnostics.

A processor **MAY** also provide a mode of operation in which [static errors](#) in parts of the stylesheet that are not evaluated can go unreported.

Note:

For example, when operating in this mode, a processor might report static errors in a template rule only if the input document contains nodes that match that template rule. Such a mode of operation can provide performance benefits when large and well-tested stylesheets are used to process source documents that might only use a small part of the XML vocabulary that the stylesheet is designed to handle.

[**DEFINITION:** An error that is not capable of detection until a source document is being transformed is referred to as a [dynamic error](#).]

When a [dynamic error](#) occurs, and is not caught using [`xsl:catch`](#), 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 a [principal result](#) or [secondary result](#) 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 [9.11 Circular Definitions](#).

There are also some [dynamic errors](#) where the specification gives a processor license to signal the error during the analysis phase even if the construct might never be executed; an example is the use of an invalid QName as a literal argument to a function such as [key](#), or the use of an invalid regular expression in the [regex](#) attribute of the [`xsl:analyze-string`](#) instruction.

A [dynamic error](#) is also signaled during the static analysis phase if the error occurs during evaluation of a [static expression](#).

The XPath specification states (see [Section 2.3.1 Kinds of Errors](#)^{XP30}) 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 or 3.0 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 [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 the value of the `select` expression of the `xsl:number` instruction (it must be a node). 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:number select="42"/>
</xsl:if>
```

On the other hand, in the following example it is not possible to determine statically whether the operand of `xsl:number` 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:number 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 `principal result` or `secondary result`, it is still possible that errors may occur in serializing that result. For example, it may be impossible to serialize the result 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 [26 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), 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](#).

Errors defined in related specifications ([XPath 3.0](#), [Functions and Operators 3.0](#) [[XSLT and XQuery Serialization](#)]) 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.

Implementations MUST use the codes defined in these specifications when signaling dynamic errors, to ensure that `xsl:catch` behaves in an interoperable way across implementations. Stylesheet authors should note, however, that there are many examples of errors where more than one rule in this specification is violated, and where the processor therefore has discretion in deciding which error code to associate with the condition: there is therefore no guarantee that different processors will always use the same error code for the same erroneous input.

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.

3 Stylesheet Structure

This section describes the overall structure of a stylesheet as a collection of XML documents.

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 [3.7 Stylesheet Element](#) and [3.8 Simplified Stylesheet Modules](#)).

XSLT [processors](#) MUST use the XML namespaces mechanism [[Namespaces in XML](#)] 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 [24.2 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:

By convention, the names of [XSLT elements](#), attributes and functions are all lower-case; they use hyphens to separate words, and they use abbreviations only if these already appear in the syntax of a related language such as XML or HTML. Names of types defined in XML Schema are regarded as single words and are capitalized exactly as in XML Schema. This sometimes leads to composite function names such as [`current-dateTime`^{FO30}](#).

3.2 Extension Attributes

[**DEFINITION:** An element from the XSLT namespace may have any attribute not from the XSLT namespace, provided that the [expanded QName](#) (see [\[XPath 3.0\]](#)) 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 [principal result](#) or any [secondary result](#) of the transformation to be different from the results that a conformant XSLT 3.0 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 described in [\[XSLT and XQuery Serialization\]](#), 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 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.

3.3 XSLT Media Type

The media type `application/xslt+xml` has been registered for XSLT stylesheet modules.

The definition of the media type is at [\[XSLT Media Type\]](#).

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](#).

3.4 Standard Attributes

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

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:default-collation`, `xsl:default-mode`, `xsl:default-validation`, `xsl:exclude-result-prefixes`, `xsl:expand-text`, `xsl:extension-element-prefixes`, `xsl:use-when`, `xsl:version`, or `xsl:xpath-default-namespace`.

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:]default-collation, [xsl:]expand-text, [xsl:]version, and [xsl:]xpath-default-namespace, 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 [3.13.1 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](#), [xsl:transform](#), and [xsl:package](#) elements only. This reflects the fact that these attributes are often used on the outermost element of the stylesheet, in which case they apply to the entire [stylesheet module](#) or [package manifest](#).

Note that the effect of these attributes does *not* extend to [stylesheet modules](#) referenced by [xsl:include](#) or [xsl:import](#) declarations, nor to packages referenced using [xsl:use-package](#).

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

[xsl:]default-collation

see [3.7.1 The default-collation Attribute](#)

[xsl:]default-mode

see [3.7.2 The default-mode Attribute](#)

[xsl:]default-validation

see [25.4 Validation](#)

[xsl:]exclude-result-prefixes

see [11.1.3 Namespace Nodes for Literal Result Elements](#)

[xsl:]expand-text

see [5.6.2 Text Value Templates](#)

[xsl:]extension-element-prefixes

see [24.2 Extension Instructions](#)

[xsl:]use-when

see [3.13.1 Conditional Element Inclusion](#)

[xsl:]version

see [3.9 Backwards Compatible Processing](#) and [3.10 Forwards Compatible Processing](#)

[xsl:]xpath-default-namespace

see [5.1.2 Unprefixed Lexical QNames in Expressions and Patterns](#)

3.5 Packages

[**DEFINITION:** An explicit **package** is represented by an [xsl:package](#) element, which will generally be the outermost element of an XML document. When the [xsl:package](#) element is not used explicitly, the entire stylesheet comprises a single implicit package.] (This specification does not preclude the [xsl:package](#) being embedded in another XML document, but it will never have any other XSLT element as an ancestor).

```
<xsl:package
  id? = id
  name? = uri
  package-version? = string
  version = decimal
  input-type-annotations? = "preserve" | "strip" | "unspecified"
  declared-modes? = boolean
  default-mode? = eqname | "#unnamed"
  default-validation? = "preserve" | "strip"
  default-collation? = uris
  extension-element-prefixes? = prefixes
  exclude-result-prefixes? = prefixes
  expand-text? = boolean
  use-when? = expression
  xpath-default-namespace? = uri >
  <!-- Content: ((xsl:expose | declarations)*)
  -->
</xsl:package>
```

[**DEFINITION:** The content of the [xsl:package](#) element is referred to as the **package manifest**].

The **version** attribute indicates the version of the XSLT language specification to which the package manifest conforms. The value **SHOULD** normally be **3.0**. If the value is numerically less than **3.0**, the content of the [xsl:package](#) element is processed using the rules for [backwards compatible behavior](#) (see [3.9 Backwards Compatible Processing](#)). If the value is numerically greater than **3.0**, it is processed using the rules for [forwards compatible behavior](#) (see [3.10 Forwards Compatible Processing](#)).

A package typically has a name, given in its **name** attribute, which **MUST** be an absolute URI. Unnamed packages are allowed, but they can only be used as the “top level” of an application; they cannot be the target of an [xsl:use-package](#) declaration in another package.

A package may have a version identifier, given in its **package-version** attribute. This is used to distinguish different versions of a package. The value of the **version** attribute, after trimming leading and trailing whitespace, **MUST** conform to the syntax given in [3.5.1 Versions of a Package](#). If no version number is specified for a package, **version 1** is assumed.

The attributes **default-collation**, **default-mode**, **default-validation**, **exclude-result-prefixes**, **expand-text**, **extension-element-prefixes**, **use-when**, **version**, and **xpath-default-namespace** are standard attributes that can appear on any XSLT element, and potentially affect all descendant elements. Their meaning is described in [3.4 Standard Attributes](#).

The package manifest contains the following elements, arbitrarily ordered:

1. Zero or more [xsl:expose](#) declarations that define the interface offered by this package to the outside world.
An [xsl:expose](#) declaration may appear only as a child of [xsl:package](#).
2. Zero or more additional [declarations](#). These are the same as the declarations permitted as children of [xsl:stylesheet](#) or [xsl:transform](#).

Some declarations of particular relevance to packages include:

- a. The [xsl:use-package](#) declaration, which declares the names and versions of the packages on which this package is dependant.
- b. The optional [xsl:global-context-item](#) element; if present this element defines constraints on the existence and type of the [global context item](#).
- c. Zero or more [xsl:include](#) and [xsl:import](#) declarations, which define additional stylesheet modules to be incorporated into this package.
- d. Zero or more ordinary [declarations](#), that is, elements that are permitted as children of [xsl:stylesheet](#) or [xsl:transform](#). One possible coding style is to include in the package manifest just a single [xsl:import](#) or [xsl:include](#) declaration as a reference to the effective top-level stylesheet module; this approach is particularly suitable when writing code that is required to run under earlier releases of XSLT as well as under XSLT 3.0. Another approach is to include the substance of the top-level stylesheet module inline within the package manifest.

Example: An example package

The following example shows a package that offers a number of functions for manipulating complex numbers. A complex number is represented as a map with two entries, the keys being 0 for the real part, and 1 for the imaginary part.

```

<xsl:package
    name="http://example.org/complex-arithmetic.xsl"
    package-version="1.0"
    version="3.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:f="http://example.org/complex-arithmetic.xsl">

    <xsl:function name="f:complex-number"
        as="map(xs:integer, xs:double)" visibility="public">
        <xsl:param name="real" as="xs:double"/>
        <xsl:param name="imaginary" as="xs:double"/>
        <xsl:sequence select="map{ 0:$real, 1:$imaginary }"/>
    </xsl:function>

    <xsl:function name="f:real"
        as="xs:double" visibility="public">
        <xsl:param name="complex" as="map(xs:integer, xs:double)"/>
        <xsl:sequence select="$complex(0)"/>
    </xsl:function>

    <xsl:function name="f:imag"
        as="xs:double" visibility="public">
        <xsl:param name="complex" as="map(xs:integer, xs:double)"/>
        <xsl:sequence select="$complex(1)"/>
    </xsl:function>

    <xsl:function name="f:add"
        as="map(xs:integer, xs:double)" visibility="public">
        <xsl:param name="x" as="map(xs:integer, xs:double)"/>
        <xsl:param name="y" as="map(xs:integer, xs:double)"/>
        <xsl:sequence select="
            f:complex-number(
                f:real($x) + f:real($y),
                f:imag($x) + f:imag($y))"/>
    </xsl:function>

    <xsl:function name="f:multiply"
        as="map(xs:integer, xs:double)" visibility="public">
        <xsl:param name="x" as="map(xs:integer, xs:double)"/>
        <xsl:param name="y" as="map(xs:integer, xs:double)"/>
        <xsl:sequence select="
            f:complex-number(
                f:real($x)*f:real($y) - f:imag($x)*f:imag($y),
                f:real($x)*f:imag($y) + f:imag($x)*f:real($y))"/>
    </xsl:function>

    <!-- etc. -->

</xsl:package>
```

A more complex package might include private or abstract functions as well as public functions; it might expose components other than functions (for example, templates or global variables), and it might contain [xsl:use-package](#) elements to allow it to call on the services of other packages.

Note:

In this example, the way in which complex numbers are represented is exposed to users of the package. It would be possible to hide the representation by declaring the types on public functions simply as `item()`; but this would be at the cost of type safety.

A package that does not itself expose any components may be written using a simplified syntax: the `xsl:package` element is omitted, and the `xsl:stylesheet` or `xsl:transform` element is now the outermost element of the stylesheet module. For compatibility reasons, all the named templates and modes declared in the package are made public. More formally, the principal stylesheet module of the [top-level package](#) may be expressed as an `xsl:stylesheet` or `xsl:transform` element, which is equivalent to the package represented by the output of the following transformation, preserving the base URI of the source:

```
<xsl:transform version="3.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:t="http://www.w3.org/1999/XSL/TransformAlias">

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

  <xsl:template match="xsl:stylesheet|xsl:transform">
    <t:package declared-modes="no">
      <xsl:copy-of select="@*"/>
      <t:expose component="mode" names="*" visibility="public"/>
      <t:expose component="template" names="*" visibility="public"/>
      <xsl:copy-of select="node()"/>
    </t:package>
  </xsl:template>
</xsl:transform>
```

The effect of the `input-type-annotations` attribute is defined in [4.4.1 Stripping Type Annotations from a Source Tree](#).

A more extensive example of a package, illustrating how components in a package can be overridden in a client package, is given in [3.5.7 Worked Example of a Library Package](#).

[3.5.1 Versions of a Package](#)

If a package has a version number, the version number must conform to the grammar:

```
PackageVersion   ::= NumericPart ( "-" NamePart )?
NumericPart     ::= IntegerLiteral ( "." IntegerLiteral )*
NamePart        ::= NCName
```

Here `IntegerLiteral`^{XP30} and `NCName` are as defined in the XPath 3.0 grammar productions of the same name (including rules on limits). Leading and trailing whitespace is ignored; no other whitespace is allowed.

Examples of valid version numbers are `2.0.5` or `3.10-alpha`.

[DEFINITION: The integer literals and the optional NamePart within the version number are referred to as the **portions** of the version number.]

Note:

This means that 1-alpha-2 is a valid version number, with two portions: 1 and alpha - 2. The second hyphen is part of the NCName, it does not act as a portion separator.

Versions are ordered. When comparing two versions:

1. Trailing zero portions (that is, any zero-valued integer that is not followed by another integer) are discarded.
2. Comparison proceeds by comparing portions pairwise from the left.
3. If both versions have the same number of portions and all portions compare equal (under the rules of the XPath eq operator using the Unicode codepoint collation), then the versions compare equal.
4. If the number of portions in the two versions V_1 and V_2 is N_1 and N_2 , with $N_1 < N_2$, and if all portions in positions 1 to N compare equal, then V_1 is less than V_2 if the portion of V_2 in position N_1 is an integer, and is greater than V_2 if this portion is an NCName. For example, 1.2 is less than 1.2.5, while 2.0 is greater than 2.0-rc1.
5. Portions are compared as follows:
 - a. If both portions are integers, they are compared using the rules of XPath value comparisons.
 - b. If both portions are NCNames, they are compared using the rules of XPath value comparisons, using the Unicode Codepoint Collation.
 - c. If one portion is an integer and the other is an NCName, the NCName comes first.

For example, the following shows a possible ordered sequence of version numbers:

```
0-rc1 < 0-rc2 < 0 < 1 = 1.0 < 1.0.2
      < 1.0.3-rc1 < 1.0.3 < 1.0.3.2 < 1.0.10
```

Note:

The version number format defined here is designed to be general enough to accommodate a variety of conventions in common use, and to allow useful semantics for matching of versions and ranges of versions, without being over-prescriptive. It is influenced by [SemVer], but is not as prescriptive, and it imposes no assumptions about backwards compatibility of packages between successive versions.

Implementations MAY impose limits on the values used in a version number (or a version range: see below). Such limits are implementation-defined. As a minimum, a processor MUST accept version numbers including:

- A numeric part containing four integers;
- Each integer being in the range 0 to 999999;
- An NCName of up to 100 characters

Dependencies between packages may specify a version range (see [3.5.2 Dependencies between Packages](#)). A version range represents a set of accepted versions. The syntax of a version range is shown below. Whitespace is permitted only where indicated, using the terminal symbol S.

```

PackageVersionRange   ::= AnyVersion | VersionRanges
AnyVersion           ::= "*"
VersionRanges        ::= VersionRange (S? "," S? VersionRange)*
VersionRange         ::= PackageVersion | VersionPrefix |
                         VersionFrom | VersionTo | VersionFromTo
VersionPrefix        ::= PackageVersion ".*"
VersionFrom          ::= PackageVersion "+"
VersionTo            ::= "to" S (PackageVersion | VersionPrefix)
VersionFromTo        ::= PackageVersion S "to" S (PackageVersion | VersionPrefix)

```

The meanings of the various forms of version range are defined below:

- The range **AnyVersion** matches any version.
- The range **VersionRanges** matches a version if any constituent **VersionRange** matches that version.
For example, 9.5.0.8, 9.6.1.2 matches those specific versions only, while 9.5.0.8, 9.6+ matches either version 9.5.0.8 or any version from 9.6 onwards.
- A range that is a **PackageVersion** matches that version only.
- The range **VersionPrefix** matches any version whose leading [portions](#) are the same as the [portions](#) in the **PackageVersion** part of the **VersionPrefix**.
For example, 1.3.* matches 1.3, 1.3.5, 1.3.10.2, and 1.3-beta (but not 1 or 1.4).

Note:

The .* indicates that additional [portions](#) may follow; it does not indicate a substring match on the final [portion](#). So 1.3.* does not match 1.35, and 3.3-beta.* does not match 3.3-beta12. Also, 3.3-beta.* does not match 3.3-beta.5: this is because the last dot is not a portion separator, but is part of the final NCName. In fact, using .* after a version number that includes an NCName portion is pointless, because an NCName portion can never be followed by further portions.

- The range **VersionFrom** matches any version that is greater than or equal to the version supplied.
For example 1.3+ matches 1.3, 1.3.2, 1.4, and 2.1 (but not 1.3-beta or 1.2). And 1.3-beta+ matches 1.3-beta, 1.3-gamma, 1.3.0, 1.4, and 8.0, but not 1.3-alpha or 1.2.
- The range **VersionTo** matches any version that is less than or equal to some version that matches the **VersionPrefix**.
For example, to 4.0 matches 1.5, 2.3, 3.8, 4.0, and 4.0-beta (but not 4.0.1), while to 3.3.* matches 1.5 or 2.0.6 or 3.3.4621, but not 3.4.0 or 3.4.0-beta.
- The range **VersionFromTo** matches any version that is greater than or equal to the starting **PackageVersion**, and less than or equal to some version that matches the **VersionPrefix**.
For example, 1 to 5 matches 1.1, 2.1, 3.1, or 5.0 (but not 5.1), while 1 to 5.* matches all of these, plus versions such as 5.7.2 (but not 6.0 or 6.0-beta). Similarly, 1.0-beta to 1.0 matches 1.0-beta, 1.0-beta.2, 1.0-gamma, and 1.0, but not 1.0-alpha or 1.0.1.

3.5.2 Dependencies between Packages

When [components](#) in one [package](#) reference components in another, the dependency of the first package on the second must be represented by an [`xsl:use-package`](#) element. This may appear in the [principal stylesheet module](#) of the first package (which may be a [package manifest](#)), or it may appear in a [stylesheet module](#) that is referenced from the [principal stylesheet module](#) via one or more [`xsl:include`](#) declarations; however it must not be referenced via [`xsl:import`](#) declarations (this is to avoid complications caused by multiple [`xsl:use-package`](#) declarations with different [import precedence](#)).

[**DEFINITION:** If a package Q contains an [`xsl:use-package`](#) element that references package P , then package Q is said to **use** package P . In this relationship package Q is referred to as the **using package**, package P as the **used package**.]

The phrase **directly uses** is synonymous with **uses** as defined above, while **directly or indirectly uses** refers to the transitive closure of this relationship.

```
<!-- Category: declaration -->
<xsl:use-package
  name = uri
  package-version? = string >
  <!-- Content: (xsl:accept | xsl:override)* -->
</xsl:use-package>
```

A [package](#) may be [used](#) by more than one other package, but the relationship **MUST NOT** be cyclic. It is possible, but by no means inevitable, that using the same package in more than one place within a stylesheet will cause static errors due to the presence of conflicting components according to the above rules. Where a package is successfully used by more than one other package, its components may be overridden in different ways by different using packages.

The [name](#) and [package-version](#) attributes together identify the used package. The value of the [package-version](#) attribute, if present, must conform to the rules for a [PackageVersionRange](#) given in [3.5.1 Versions of a Package](#); if omitted the value $*$ is assumed, which matches any version. The used package must have a name that is an exact match for the name in the [name](#) attribute (using codepoint comparison), and its explicit or implicit [package-version](#) must match the version range given in the [package-version](#) attribute.

This specification does not define how the implementation locates a package given its name and version. If several matching versions of a package are available, it does not define which of them is chosen. Nor does it define whether this process locates source code or some other representation of the package contents. Such mechanisms are [implementation-defined](#). Use of the package name as a dereferenceable URI is **NOT RECOMMENDED**, because the intent of the packaging feature is to allow a package to be distributed as reusable code and therefore to exist in many different locations.

[ERR XTSE3000] It is a [static error](#) if no package matching the package name and version specified in an [`xsl:use-package`](#) declaration can be located.

[ERR XTSE3005] It is a [static error](#) if a package is dependent on itself, where package A is defined as being dependent on package B if A contains an [`xsl:use-package`](#) declaration that references B , or if A contains an [`xsl:use-package`](#) declaration that references a package C that is itself dependent on B .

[ERR XTSE3008] It is a [static error](#) if an [`xsl:use-package`](#) declaration appears in a [stylesheet module](#) that is not in the same [stylesheet level](#) as the [principal stylesheet module](#) of the [package](#).

Note:

Depending on the implementation architecture, there may be a need to locate used packages both during static analysis (for example, to get information about the names and type signatures of the components exposed by the used package), and also at evaluation time (to link to the implementation of these components so they can be invoked). A failure to locate a package may cause an error at either stage.

The [xsl:accept](#) and [xsl:override](#) elements are used to modify the visibility or behavior of components acquired from the used package; they are described in [3.5.3.2 Accepting Components](#) below.

Note:

It is not intrinsically an error to have two [xsl:use-package](#) declarations that identify the same package (or different versions of the same package). This has the same effect as having two declarations that identify packages with different names but identical content. In most cases it will result in an error ([see [ERR_XTSE3050](#)]) due to the presence of multiple components with the same name; but no error would occur, for example, if the used package is empty, or if the two [xsl:use-package](#) declarations use [xsl:accept](#) to accept non-overlapping subsets of the components in the used package.

[3.5.3 Named Components in Packages](#)

This section discusses the use of named components in packages.

The components which can be declared in one package and referenced in another are: [functions](#), [named templates](#), [attribute sets](#), [modes](#), and [global variables](#) and [parameters](#).

In addition, [keys](#) and [accumulators](#) are classified as named components because they can contain references to components in another package, even though they cannot themselves be referenced from outside the package.

Named and unnamed [modes](#) come within the scope of this section, but there are differences noted in [3.5.4 Overriding Template Rules from a Used Package](#).

Not all [declarations](#) result in [components](#):

- Named [declarations](#) that can neither be referenced from outside their containing package, nor can contain references to components in other packages (examples are [xsl:output](#), [xsl:character-map](#), and [xsl:decimal-format](#)) are not considered to be components and are therefore outside the scope of this section.
- Some declarations, such as [xsl:decimal-format](#) and [xsl:strip-space](#), declare aspects of the processing context which are not considered to be components as defined here.
- [Template rules \(xsl:template\)](#) with a [match](#) attribute are also not considered to be components for the purposes of this section, which is concerned only with components that are bound by name. However, when an [xsl:template](#) has both a [match](#) attribute and a [name](#) attribute, then it establishes both a template rule and a [named template](#), and in its role as a named template it comes within the scope of this discussion.
- A named declaration, for example a named template, a function, or a global variable, may be overridden within the same package by another like-named declaration having higher [import precedence](#). When a

declaration is overridden in this way it cannot be referenced by name either from within its containing package or from outside that package.

- In the case of [xsl:attribute-set](#) and [xsl:key](#) declarations, several declarations combine to form a single component.

The section is largely concerned with details of the rules that affect references from one component to another by name, whether the components are in the same package or in different packages. The rules are designed to meet a number of requirements:

- A component defined in one package can be overridden by a component in another package, provided the signatures are type-compatible.
- The author of a package can declare whether the components in the package are public or private (that is, whether or not they can be used from outside the package) and whether they are final, overridable, or abstract (that is whether they can or must be overridden by the using package).
- Within an application, two packages can make use of a common library and override its components in different ways.
- Visibility of components can be defined either as part of the declaration of the component, or in the package manifest.
- An application that wishes to make use of a [library package](#) can be selective about which components from the library it acquires, perhaps to avoid name clashes between components acquired from different libraries.

[**DEFINITION:** The term **component** is used to refer to any of the following: a [stylesheet function](#), a [named template](#), a [mode](#), an [accumulator](#), an [attribute set](#), a [key](#), [global variable](#), or a [mode](#).]

[**DEFINITION:** The **symbolic identifier** of a [component](#) is a composite name used to identify the component uniquely within a package. The symbolic identifier comprises the kind of component (stylesheet function, named template, accumulator, attribute set, global variable, key, or mode), the [expanded QName](#) of the component (namespace URI plus local name), and in the case of stylesheet functions, the [arity](#).]

Note:

In the case of the [unnamed mode](#), the expanded QName of the component may be considered to be some system-allocated name different from any user-defined mode name.

[**DEFINITION:** Two [components](#) are said to be **homonymous** if they have the same [symbolic identifier](#).]

Every [component](#) has a [declaration](#) in some [stylesheet module](#) and therefore within some [package](#). In the case of [attribute sets](#) and [keys](#), there may be several declarations. The declaration is an element in an XDM tree representing the stylesheet module. Declarations therefore have identity, based on XDM node identity.

[**DEFINITION:** The **declaring package** of a [component](#) is the package that contains the declaration (or, in the case of [xsl:attribute-set](#) and [xsl:key](#), multiple declarations) of the component.]

When a [component](#) declared in one [package](#) is made available in another, the using package will contain a separate component that can be regarded as a modified copy of the original. The new component shares the same [symbolic identifier](#) as the original, and it has the same [declaration](#), but it has other properties such as its [visibility](#) that may differ from the original.

[**DEFINITION:** A component declaration results in multiple components, one in the package in which the declaration appears, and potentially one in each package that uses the declaring package, directly or indirectly, subject to the visibility of the component. Each of these multiple components has the same [declaring package](#), but each has a different **containing package**. For the original component, the declaring package and the containing package are the same; for a copy of a component made as a result of an [xsl:use-package](#) declaration, the declaring package will be the original package, and the containing package will be the package in which the [xsl:use-package](#) declaration appears.]

Note:

Within this specification, we generally use the notation C_P for a component named C whose declaring package and containing package are both P; and the notation C_{PQ} for a component whose containing package is P and whose declaring package is Q (that is, a component in P that is derived from a component C_Q in the used package Q).

The properties of a [component](#) are as follows:

- The original [declaration](#) of the component.
- The [package](#) to which the component belongs (called its **containing** package, not to be confused with the [declaring package](#)).
- The [symbolic identifier](#) of the component.
- The [visibility](#) of the component, which determines the way in which the component is seen by other components within the same package and within using packages. This is one of **public**, **private**, **abstract**, **final**, or **hidden**. The visibility of components is discussed further in [3.5.3.1 Visibility of Components](#).
- A set of bindings for the [symbolic references](#) in the component. The way in which these bindings are established is discussed further in [3.5.3.5 Binding References to Components](#).

Note:

When a function F defined in a package P is acquired by two using packages Q and R, we may think of P, Q, and R as all providing access to the “same” function. The detailed semantics, however, demand an understanding that there is one function declaration, but three components. The three components representing the function F within packages P, Q, and R have some properties in common (the same symbolic identifier, the same declaration), but other properties (the visibility and the bindings of symbolic references) that may vary from one of these components to another.

[**DEFINITION:** The [declaration](#) of a component includes constructs that can be interpreted as references to other [components](#) by means of their [symbolic identifiers](#). These constructs are generically referred to as **symbolic references**. Examples of constructs that give rise to symbolic references are the name attribute of [xsl:call-template](#); the [xsl:]use-attribute-sets attribute of [xsl:copy](#), [xsl:element](#), and [literal result elements](#); the explicit or implicit mode attribute of [xsl:apply-templates](#); XPath variable references referring to global variables; XPath static function calls (including partial function applications) referring to [stylesheet functions](#); and named function references (example: my:f#1) referring to stylesheet functions.]

Symbolic references exist as properties of the [declaration](#) of a [component](#). The [symbolic identifier](#) being referred to can be determined straightforwardly from the syntactic form and context of the reference: for example, the instruction `<xsl:value-of select="f:price($o)" xmlns:f="http://f.com/">` contains a symbolic reference to a function with expanded name `{http://f.com/}price` and with arity=1. However, because there may be several (homonymous) function components with this symbolic identifier, translating this symbolic reference into a reference to a specific component (a process called “binding”) is less straightforward, and is described in the text that follows.

The process of assembling a stylesheet from its constituent packages is primarily a process of binding these symbolic references to actual components. Within any [component](#) whose [declaration](#) is D , there is a set of bindings; each binding is an association between a [symbolic reference](#) in D and a [component](#) whose [symbolic identifier](#) matches the outward reference. Outward references for which a component C contains a binding are said to be **bound** in C ; those for which C contains no binding are said to be **unbound**.

For example, suppose that in some package Q , function A calls B , which in turn calls C , and that B is **private**. Now suppose that in some package P which uses Q , C is overridden. The effect of the binding process is that P will contain three components corresponding to A , B , and C , which we might call A_P , B_P , and C_P . The [declarations](#) of A_P and B_P are in package Q , but the declaration of C_P is in P . The internal visibility of B_P will be **hidden** (meaning that it cannot be referenced from within P), and B_P will contain a binding for the component C_P that corresponds to the outward reference from B to C . The effect is that when A calls B and B calls C , it is the overriding version of C that is executed.

In another package R that uses Q without overriding C , there will be three different components A_R , B_R , and C_R . This time the declaration of all three components is in the original package Q . Component B_R will contain a binding to C_R , so in this package, the original version of C is executed. The fact that one package P overrides C thus has no effect on R , which does not override it.

The binding process outlined above is described in more detail in [3.5.3.5 Binding References to Components](#).

Template rules are not components in their own right; unlike named templates, they are never referenced by name. Component references within a template rule (for example, references to functions, global variables, or named templates) are treated as occurring within the component that represents the containing mode. This includes component references within the match patterns of template rules. If a template rule lists several modes, it is treated as if there were multiple template rules one in each mode.

An [xsl:apply-templates](#) instruction with no `mode` attribute is treated as a reference to the default mode defined for that [instruction](#) (see [3.7.2 The default-mode Attribute](#)), which in turn defaults to the [unnamed mode](#). An implicit reference to the unnamed mode is treated in the same way as any other [symbolic reference](#). Note that there is an unnamed mode in every package, and the unnamed mode always has private visibility.

Where an [xsl:template](#) element has both a `name` and a `match` attribute, it is treated as if there were two separate [xsl:template](#) elements, one with a `name` attribute and one with a `match` attribute.

[Keys](#) and [accumulators](#) behave rather differently from other components. Their visibility is always private, which means they can only be used within their declaring package. In addition, the component binding is generally made dynamically rather than statically, by virtue of a string passed as an argument to the function [key](#), [accumulator-before](#), or [accumulator-after](#). (In the case of accumulators, there can also be static references: see the `use-accumulators` attribute of [xsl:source-document](#), [xsl:merge-source](#), and [xsl:mode](#).) However, outward references from key definitions and [accumulators](#) to other components (such as global variables and functions)

behave in the same way as component references contained in any other private component, in that they may be rebound to an overriding declaration of the target component.

3.5.3.1 *Visibility of Components*

[**DEFINITION:** The **visibility** of a component is one of: **private**, **public**, **abstract**, **final**, or **hidden**.]

The meanings of these visibility values is as follows:

public

The component can be referenced from other components in this package or in any using package; it can be overridden by a different component in any using package.

private

The component can be referenced from other components in this package; it cannot be referenced or overridden within a using package.

abstract

The component can be referenced from other components in this package or in any using package; in a using package it can either remain abstract or be overridden by a different component.

final

The component can be referenced from other components in this package or in any using package; it cannot be overridden by a different component in any using package.

hidden

The component cannot be referenced from other components in this package; it cannot be referenced or overridden within a using package.

Note:

The visibility of a component in a package P primarily affects how the component can be used in other packages, specifically, packages that use P . There is one exception: if the visibility is **hidden**, it also affects how the component can be used within P .

When a component is declared within a particular package, its visibility, which affects how it can be used in other (using) packages, depends on two factors: the value of the **visibility** declaration on the declaration itself (if present), and the rules given in the xsl:expose declarations of the package manifest.

The xsl:function, xsl:template, xsl:attribute-set, xsl:variable, and xsl:mode declarations each have an optional **visibility** attribute. The value is one of **private**, **public**, **abstract**, or **final** (never **hidden**). In the case of an xsl:param element there is no explicit **visibility** attribute; rather the declaration has the implicit attribute **visibility="public"**.

Any xsl:expose declarations that appear as children of xsl:package define the visibility of components whose declaration has no explicit **visibility** attribute, and can also be used to reduce the visibility of components where this attribute is present.

```

<xsl:expose
  component = "template" | "function" | "attribute-set" | "variable" | "mode" |
  "*"
  names = tokens
  visibility = "public" | "private" | "final" | "abstract" />

```

The [xsl:expose](#) element allows the [visibility](#) of selected components within a package to be defined.

The components in question are identified using their [symbolic identifiers](#). The `component` attribute defines the kind of component that is selected. The value `*` means “all component kinds”; in this case the value of the `names` attribute must be a [Wildcard^{XP30}](#).

An [xsl:expose](#) declaration has no effect on the [unnamed mode](#), which is always private to a package.

The `names` attribute selects a subset of these components by name (and in the case of functions, arity); its value is a whitespace-separated sequence of tokens each of which is either a [NameTest^{XP30}](#) or a [NamedFunctionRef^{XP30}](#). (Examples are `*`, `p:*`, `*:local`, `p:local`, and `p:local#2`.)

The value may be a [NamedFunctionRef](#) only in the case of stylesheet functions, and distinguishes functions with the same name and different arity.

The visibility of a named template, function, variable, attribute set, or mode declared within a package is the first of the following that applies, subject to consistency constraints which are defined below:

1. The visibility of a variable declared using an [xsl:param](#) element is always `public`. No [xsl:expose](#) element ever matches an [xsl:param](#) component.

Note:

Attempting to match an [xsl:param](#) with an explicit EQName will therefore always give an error, while using a wildcard has no effect.

2. If the package manifest contains an [xsl:expose](#) element that matches this component by virtue of an explicit EQName or [NamedFunctionRef](#) (that is, not by virtue of a wildcard match), then the value of the `visibility` attribute of the last such [xsl:expose](#) element in document order (call this the **explicit exposed visibility**).
3. If the declaration of the component has a `visibility` attribute, then the value of this attribute (call this the **declared visibility**).
4. If the package manifest contains an [xsl:expose](#) element that matches this component by virtue of a wildcard match that specifies either the namespace part of the component name or the local part of the name (for example, `prefix:*` or `*:local` or `Q{uri}*`), then the value of the `visibility` attribute of the last such [xsl:expose](#) element in document order.
5. If the package manifest contains an [xsl:expose](#) element that matches this component by virtue of a wildcard match that matches all names (that is, `*`), then the value of the `visibility` attribute of the last such [xsl:expose](#) element in document order.
6. Otherwise, `private`.

Note:

In the above rules, no distinction is made between declarations that specify a specific component kind, and those that specify `component="*"`. If both match, the value of the `component` attribute plays no role in deciding which declaration wins.

If both a declared visibility and an explicit exposed visibility exist for the same component, then as mentioned above, they must be consistent. This is determined by reference to the following table, where the entry N/P means “not permitted”. (In cases where the combination is permitted, the actual visibility is always the same as the visibility determined by [`xsl:expose`](#).)

Relationship of Exposed Visibility to Potential Visibility

Explicit exposed visibility	Declared visibility			
	public	private	final	abstract
public	public	N/P	N/P	N/P
private	private	private	private	N/P
final	final	N/P	final	N/P
abstract	N/P	N/P	N/P	abstract

[ERR XTSE3010] It is a [static error](#) if the explicit exposed visibility of a component is inconsistent with its declared visibility, as defined in the above table. (This error occurs only when the component declaration has an explicit `visibility` attribute, and the component is also listed explicitly by name in an [`xsl:expose`](#) declaration.)

[ERR XTSE3020] It is a [static error](#) if a token in the `names` attribute of [`xsl:expose`](#), other than a wildcard, matches no component in the containing package.

[ERR XTSE3022] It is a [static error](#) if the `component` attribute of [`xsl:expose`](#) specifies * (meaning all component kinds) and the `names` attribute is not a wildcard.

Note:

There is no ambiguity, and no error, if several tokens within the same [`xsl:expose`](#) element match the same component.

If the visibility of a component as established by the above rules is `abstract`, then the component must have a declared visibility of `abstract`.

Note:

In other words, the [`xsl:expose`](#) declaration cannot be used to make a component abstract unless it was declared as abstract to start with.

[ERR XTSE3025] It is a [static error](#) if the effect of an [`xsl:expose`](#) declaration would be to make a component [`abstract`](#), unless the component is already [`abstract`](#) in the absence of the [`xsl:expose`](#) declaration.

For a component accepted into a package P from another package Q , the [`visibility`](#) of the component in P (which primarily affects how it can be used in a package R that uses P) depends on the visibility declared in the relevant [`xsl:accept`](#) or [`xsl:override`](#) element in P (see [3.5.3.2 Accepting Components](#)); this in turn has a default that depends on the visibility of the corresponding component in Q . In this case the visibility is unaffected by any [`xsl:expose`](#) declaration in P .

[3.5.3.2 Accepting Components](#)

When a package P uses a package Q , by virtue of an [`xsl:use-package`](#) element in the [`package manifest`](#) of P , then P will contain a [`component`](#) corresponding to every component in Q . The [`visibility`](#) of the component within P depends on the [`visibility`](#) of the component in Q , optionally modified by two elements that may appear as children of the [`xsl:use-package`](#) element, namely [`xsl:accept`](#) and [`xsl:override`](#).

For every component C_Q in package Q that is not matched by any [`xsl:override`](#) or [`xsl:accept`](#) element in the package manifest of P , there will be a corresponding component C_P in package P that has the same [`symbolic identifier`](#) and [`declaration`](#) as C_Q .

If C_Q is an [`xsl:param`](#) component, then the [`visibility`](#) of C_P is [`public`](#).

In other cases, the [`visibility`](#) of C_P depends on the [`visibility`](#) of C_Q , as defined by the following table:

Visibility of Components in Used and Using Packages

Visibility in used package C_Q	Visibility in using package C_P
public	private
final	private
private	hidden
hidden	hidden
abstract	hidden

Note:

The effect of these rules is as follows:

- Components that are public or final in the used package *Q* become private in the using package *P*. This means that they can be referenced within *P* but are not (by default) visible within a package *R* that uses *P*.
- Components that are private or hidden in the used package *Q* become hidden in the using package *P*. This means that they cannot be referenced within *P*; but if they contain references to components that are overridden in *P*, the hidden component's references are bound to the overriding components in *P*.
- Components that are abstract in the used package *Q* become hidden in the using package *P*. The hidden component in this case raises a dynamic error if it is invoked. Such an invocation cannot originate within *P*, because the component is not visible within *P*; but it can occur if a public component in *Q* is invoked, which in turn invokes the abstract component.

```
<xsl:accept
  component = "template" | "function" | "attribute-set" | "variable" | "mode" |
  "*"
  names = tokens
  visibility = "public" | "private" | "final" | "abstract" | "hidden" />
```

The `xsl:accept` element has very similar syntax and semantics to `xsl:expose`. Whereas `xsl:expose` allows a package to restrict the visibility of its own components to other (using) packages, `xsl:accept` allows a package to restrict the visibility of components exposed by a package that it uses. This may be necessary if, for example, it uses two different packages whose component names conflict. It may also simply be good practice if the package author knows that only a small subset of the functionality of a used package is required.

The rules for determining whether an `xsl:accept` element matches a particular component, and for which element to use if there are several matches, are the same as the rules for the `xsl:expose` element.

No `xsl:accept` element ever matches a variable declared using `xsl:param`.

Note:

Attempting to match an `xsl:param` with an explicit QName will therefore always give an error, while using a wildcard has no effect.

[ERR XTSE3030] It is a `static error` if a token in the `names` attribute of `xsl:accept`, other than a wildcard, matches no component in the used package.

[ERR XTSE3032] It is a `static error` if the `component` attribute of `xsl:accept` specifies * (meaning all component kinds) and the `names` attribute is not a wildcard.

In the absence of a matching `xsl:override` element (see [3.5.3.3 Overriding Components from a Used Package](#)), the `visibility` of a component that matches an `xsl:accept` element depends both on the `visibility` attribute of the best-matching `xsl:accept` element and on the `visibility` of the corresponding component in the used package, according to the following table. In this table the entry “N/P” means “not permitted”.

Relationship of the Visibility given in `xsl:accept` to Visibility in the Used Package

Visibility in <code>xsl:accept</code> element	Visibility in used package			
	public	private	final	abstract
public	public	N/P	N/P	N/P
private	private	N/P	private	N/P
final	final	N/P	final	N/P
abstract	N/P	N/P	N/P	abstract
hidden	hidden	N/P	hidden	hidden

[ERR XTSE3040] It is a [static error](#) if the visibility assigned to a component by an `xsl:accept` element is incompatible with the visibility of the corresponding component in the used package, as defined by the above table, unless the token that matches the component name is a wildcard, in which case the `xsl:accept` element is treated as not matching that component.

[ERR XTSE3050] It is a [static error](#) if the `xsl:use-package` elements in a [package manifest](#) cause two or more [homonymous](#) components to be accepted with a visibility other than `hidden`.

Conflicts between the components accepted from used packages and those declared within the package itself are handled as follows:

1. If the conflict is between two components both declared within the package itself, then it is resolved by the rules relating to [import precedence](#) defined for each kind of component.
2. If the conflict is between two components both accepted from used packages, or between a component declared within the package and an accepted component, then a static error occurs.
3. If a component is explicitly accepted from a used package (by name, rather than by a matching wildcard), and if the same component is the subject of an `xsl:override` declaration, then a static error occurs (see below). There is no conflict, however, if a component declared within `xsl:override` also matches a wildcard in an `xsl:accept` element.

[ERR XTSE3051] It is a [static error](#) if a token in the `names` attribute of `xsl:accept`, other than a wildcard, matches the symbolic name of a component declared within an `xsl:override` child of the same `xsl:use-package` element.

Where the used package *Q* contains a component whose visibility is `abstract`, the using package *P* has three options:

1. *P* can accept the component with `visibility="abstract"`. In this case *P* can contain references to the component, but invocation via these references will fail unless a non-abstract overriding component has been supplied in some package *R* that (directly or indirectly) uses *P*.
2. *P* can accept the component with `visibility="hidden"`. In this case *P* cannot contain references to the component, and invocation via references in *Q* will always fail with a dynamic error. This is the default if *P* does not explicitly accept or override the component.
3. *P* can provide a concrete implementation of the component within an `xsl:override` element.

Any invocation of the absent component (typically from within its declaring package) causes a dynamic error, as if the component were overridden by a component that unconditionally raises a dynamic error.

[ERR XTDE3052] It is a [dynamic error](#) if an invocation of an abstract component is evaluated.

Note:

This can occur when a public component in the used package invokes an abstract component in the used package, and the using package provides no concrete implementation for the component in an [`xsl:override`](#) element.

Note:

To override a component accepted from a used package, the overriding declaration must appear as a child of the [`xsl:override`](#) element.

Note:

There is no rule that prevents a function (say) being declared in the using package with the same name as a `private` function in the used package. This does not create a conflict, since all references in the used package are bound to one function and all those in the using package are bound to another.

3.5.3.3 [Overriding Components from a Used Package](#)

[**DEFINITION:** A component in a using package may **override** a component in a used package, provided that the [visibility](#) of the component in the used package is either `abstract` or `public`. The overriding declaration is written as a child of the [`xsl:override`](#) element, which in turn appears as a child of [`xsl:use-package`](#).]

```
<xsl:override>
  <!-- Content: (xsl:template | xsl:function | xsl:variable | xsl:param |
    xsl:attribute-set)* -->
</xsl:override>
```

Note:

This mechanism is distinct from the mechanism for overriding declarations within the same package by relying on [import precedence](#). It imposes stricter rules: the overriding component is required to be type-compatible with the component that it overrides.

If the used package Q contains a [component](#) C_Q and the [`xsl:use-package`](#) element contains an [`xsl:override`](#) element which contains a declaration D whose [symbolic identifier](#) matches the symbolic identifier of C_Q , then the using package P will contain a component C_P whose declaration is D , whose symbolic identifier is that of D , and whose [visibility](#) is equal to the value of the `visibility` attribute of D , or `private` if this is absent, except in the case of [`xsl:param`](#), which is implicitly `public`.

The using package P will also contain a component C_{PQ} whose body is the same as the body of C_Q and whose [visibility](#) is hidden. This component is used as the target of a binding for the symbolic reference [xsl:original](#) described below.

Other than its appearance as a child of [xsl:override](#), the overriding declaration is a normal [xsl:function](#), [xsl:template](#), [xsl:variable](#), [xsl:param](#), or [xsl:attribute-set](#) element. In the case of [xsl:variable](#) and [xsl:param](#), the variable that is declared is a [global variable](#).

The rules in the remainder of this section apply to components having a [name](#) attribute (**named components**). The only element with no [name](#) attribute that can appear as a child of [xsl:override](#) is an [xsl:template](#) declaration having a [match](#) attribute (that is, a [template rule](#)). The rules for overriding of template rules appear in [3.5.4 Overriding Template Rules from a Used Package](#). If an [xsl:template](#) element has both a [name](#) attribute and a [match](#) attribute, then it defines both a named component and a template rule, and both sections apply.

[ERR XTSE3055] It is a [static error](#) if a component declaration appearing as a child of [xsl:override](#) is [homonymous](#) with any other declaration in the using package, regardless of [import precedence](#), including any other overriding declaration in the package manifest of the using package.

Note:

When an attribute set is overridden, the overriding attribute set must be defined using a single [xsl:attribute-set](#) element. Attribute sets defined in different packages are never merged by virtue of having the same name, though they may be merged explicitly by using the [use-attribute-sets](#) attribute.

[ERR XTSE3058] It is a [static error](#) if a component declaration appearing as a child of [xsl:override](#) does not match (is not [homonymous](#) with) some component in the used package.

[ERR XTSE3060] It is a [static error](#) if the component referenced by an [xsl:override](#) declaration has [visibility](#) other than [public](#) or [abstract](#).

A package is executable if and only if it contains no [component](#) whose [visibility](#) is [abstract](#). A package that is not executable is not a [stylesheet](#), and therefore cannot be nominated as the stylesheet to be used when initiating a transformation.

Note:

In other words, if a component is declared as abstract, then some package that uses the declaring package or that component directly or indirectly must override that component with one that is not abstract. It is not necessary for the override to happen in the immediately using package.

[ERR XTSE3070] It is a [static error](#) if the signature of an overriding component is not [compatible](#) with the signature of the component that it is overriding.

[DEFINITION: The signatures of two [components](#) are **compatible** if they present the same interface to the user of the component. The additional rules depend on the kind of component.]

Compatibility is only relevant when comparing two components that have the same [symbolic identifier](#). The compatibility rules for each kind of component are as follows:

- Two attribute sets with the same name are compatible if and only if they satisfy the following rule:
 1. If the overridden attribute set specifies `streamable="yes"` then the overriding attribute set also specifies `streamable="yes"`.
- Two functions with the same name and arity are compatible if and only if they satisfy all the following rules:
 1. The declared types of the arguments (defaulting to `item(*)`) are pairwise identical.
 2. The declared return types (defaulting to `item(*)`) are identical.
 3. The effective value of the `new-each-time` attribute on the overriding function is the same as its value on the overridden function.
 4. If the overridden function specifies `streamable="yes"` then the overriding function also specifies `streamable="yes"`, and in addition, it has the same posture and sweep as the function that it overrides.
- Two named templates with the same name are compatible if and only if they satisfy all the following rules:
 1. Their return types are identical.
 2. For every non-tunnel parameter on the overridden template, there is a non-tunnel parameter on the overriding template that has the same name, an identical required type, and the same effective value for the required attributes.
 3. For every tunnel parameter P on the overridden template, if there is a parameter Q on the overriding template that has the same name as P then Q is also a tunnel parameter, and P and Q have identical required types.
 4. Any parameter on the overriding template for which there is no corresponding parameter on the overridden template specifies `required="no"`.
 5. The two templates have equivalent `xsl:context-item` children, where equivalence means that the `use` attributes are the same and the required types are identical; an absent `xsl:context-item` is equivalent to one that specifies `use="optional"` and `as="item()"`.
- Two variables (including parameters) with the same name are compatible if and only if they satisfy all the following rules:
 1. Their declared types are identical.

Note:

A variable may override a parameter or vice-versa, and the initial value may differ.

Because static variables and parameters are constrained to have visibility `private`, they cannot be overridden in another package. The compatibility rules therefore do not arise. The reason that such variables cannot be overridden is that they are typically used during stylesheet compilation (for example, in `[xsl:]use-when` expressions and shadow attributes) and it is a design goal that packages should be capable of independent compilation.

[**DEFINITION:** Types S and T are considered **identical** for the purpose of these rules if and only if $\text{subtype}(S, T)$ and $\text{subtype}(T, S)$ both hold, where the subtype relation is defined in [Section 2.5.6.1 The judgement `subtype\(A, B\)`](#)^{XP30}.]

Note:

- One consequence of this rule is that two plain union types are considered identical if they have the same set of member types, even if the union types have different names or the ordering of the member types is different.

Consider a function that accepts an argument whose declared type is a union type with member types `xs:double` and `xs:decimal`, in that order (we might write this as `union(xs:double, xs:decimal)`). Using the same notation, this can be overridden by a function that declares the argument type as `union(xs:decimal, xs:double)`. This does not affect type checking: a function call that passes the type checking rules with one signature will also pass the type checking rules with the other. It does however affect the way that the function conversion rules work: a call that passes the `xs:untypedAtomic` value "93.7" (or an untyped node with this as its string value) will be converted to an `xs:decimal` in one case and an `xs:double` in the other.

- While this rule may appear formal, it is not as straightforward as might be supposed, because the subtype relation in XPath has a dependency on the “Type derivation OK (Simple)” relation in XML Schema, which itself appeals to a judgement as to whether the two type definitions being compared “are the same type definition”. Both XSD 1.0 and XSD 1.1 add the note “The wording of [this rule] appeals to a notion of component identity which is only incompletely defined by this version of this specification.” However, they go on to say that component identity is well defined if the components are named simple type definitions, which will always apply in this case. For named atomic types, the final result of these rules is that two atomic types are identical if and only if they have the same name.

Modes are not overridable, so the `xsl:mode` declaration cannot appear as a child of `xsl:override`.

3.5.3.4 Referring to Overridden Components

Within the declaration of an overriding named `component` (that is, a component whose declaration is a child of `xsl:override`, and has a `name` attribute), where the overridden component has public `visibility`, it is possible to use the name `xsl:original` as a symbolic reference to the overridden component. More specifically:

- Within a `named template` appearing as a child of `xsl:override`, the name `xsl:original` may appear as the value of the `name` attribute of `xsl:call-template`: for example, `<xsl:call-template name="xsl:original"/>`.
- Within a `stylesheet function` appearing as a child of `xsl:override`, the static context for contained XPath expressions (other than `static expressions`) is augmented as follows: the **statically known function signatures** includes a mapping from the name `xsl:original` to the signature of the overridden function (which is the same as the signature of the overriding function). This means that the name `xsl:original` can be used in static function calls, including calls that use partial function application (where one of the arguments is given as "?"), and also in named function references. For example: `xsl:original($x)`, `xsl:original($x, ?)`, `xsl:original#2`.

Note:

The result of calling `function-name(xsl:original#2)` is the name of the overridden function, not `xsl:original`.

Neither `xsl:original`, nor the overridden function, is added to the **named functions** component of the dynamic context for XPath expressions within the overriding function. This means that any attempt to bind the function name `xsl:original` dynamically (for example using `function-lookup`^{FO30}, or `function-available`, or `xsl:evaluate`) will fail, and any attempt to bind the name of the overriding/overridden function dynamically will return the overriding function.

- Within a `global variable` or parameter appearing as a child of `xsl:override`, the static context for contained XPath expressions (other than `static expressions`) is augmented as follows: the **in-scope variables** includes a mapping from the name `xsl:original` to the declared type of the overridden variable or parameter (which is the same as the type of the overriding global variable or parameter).
- Within an `attribute set` appearing as a child of `xsl:override`, any `[xsl:]use-attribute-sets` attribute (whether on the `xsl:attribute-set` element itself, or on any descendant element) may include the name `xsl:original` as a reference to the overridden attribute set.

Within the overriding component C_P , the `symbolic reference` `xsl:original` is bound to the hidden component C_{PQ} described earlier, whose body is that of the component C_Q in the used package.

[ERR XTSE3075] It is a `static error` to use the component reference `xsl:original` when the overridden component has `visibility="abstract"`.

Modes are not overridable, so the name `xsl:original` cannot be used to refer to a `mode` (for example in the `mode` attribute of `xsl:apply-templates`).

Note:

In the case of variables, templates, and attribute sets, the invocation of the overridden component can occur only within the lexical scope of the overriding component. With functions, however, there is greater flexibility. The overriding component can obtain a reference to the overridden component in the form of a function item, and can export this value by passing it to other functions or returning it in its result. A dynamic invocation of this function item (and hence, of the overridden function) can thus occur anywhere.

3.5.3.5 *Binding References to Components*

[**DEFINITION:** The process of identifying the `component` to which a `symbolic reference` applies (possibly chosen from several `homonymous` alternatives) is called **reference binding**.]

The process of `reference binding` in the presence of overriding declarations is best illustrated by an example. The formal rules follow later in the section.

Example: Binding References to Named Components

Consider a package Q defined as follows:

```
<xsl:package name="Q"
    version="3.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:variable name="A" visibility="final" select="$B + 1"/>
    <xsl:variable name="B" visibility="private" select="$C * 2"/>
    <xsl:variable name="C" visibility="public" select="22"/>
</xsl:package>
```

(The process is illustrated here using variables as the components, but the logic would be the same if the example used functions, named templates, or attribute sets.)

There are three components in this package, and their properties are illustrated in the following table. (The ID column is an arbitrary component identifier used only for the purposes of this exposition.)

Components in the above Package and their Properties

ID	Symbolic Name	Declaring Package	Containing Package	Visibility	Body	Bindings
A_Q	variable A	Q	Q	final	$\$B + 1$	$\$B \rightarrow B_Q$
B_Q	variable B	Q	Q	private	$\$C * 2$	$\$C \rightarrow C_Q$
C_Q	variable C	Q	Q	public	22	none

Now consider a package P that uses Q , and that overrides one of the variables declared in Q :

```
<xsl:package name="P"
    version="3.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:use-package name="Q">
        <xsl:override>
            <xsl:variable name="C" visibility="private" select="$xsl:original + 3"/>
        </xsl:override>
    </xsl:use-package>

    <xsl:template name="T" visibility="public">
        <xsl:value-of select="$A"/>
    </xsl:template>
</xsl:package>
```

Package P has five components, whose properties are shown in the following table:

Components in the above Package and their Properties

ID	Symbolic Name	Declaring Package	Containing Package	Visibility	Body	Bindings
A_{PQ}	variable A	Q	P	final	$\$B + 1$	$\$B \rightarrow B_{PQ}$
B_{PQ}	variable B	Q	P	hidden	$\$C * 2$	$\$C \rightarrow C_P$
C_{PQ}	variable C	Q	P	hidden	22	none
C_P	variable C	P	P	private	$\$xsl:original + 3$	$\$xsl:original \rightarrow C_{PQ}$
T_P	template T	P	P	public	<code>value-of select="\$A"</code>	$\$A \rightarrow A_{PQ}$

The effect of these bindings is that when template T is called, the result is 51. This is why:

1. The result of T is the value of A_{PQ} .
2. The value of A_{PQ} is the value of B_{PQ} plus 1.
3. The value of B_{PQ} is the value of C_P times 2.
4. The value of C_P is the value of C_{PQ} plus 3.
5. The value of C_{PQ} is 22.
6. So the final result is $((22 + 3) * 2) + 1$

In this example, the components of P are established in three different ways:

1. Components A_{PQ} , B_{PQ} , and C_{PQ} are modified copies of the corresponding component A_Q , B_Q , and C_Q in the used package Q . The properties of these components are modified as follows:
 - a. The [symbolic identifier](#), [declaring package](#), and body are unchanged.
 - b. The [containing package](#) is changed to P .
 - c. The [visibility](#) is changed according to the rules in [3.5.3.2 Accepting Components](#): in particular, `visibility="private"` changes to `visibility="hidden"`.
 - d. The references to other components are rebound as described in this section.
2. Component C_P is the overriding component. Its properties are exactly as if it were declared as a top-level component in P (outside the [xsl:use-package](#) element), except that (a) it must adhere to the constraints on overriding components (see [3.5.3.3 Overriding Components from a Used Package](#)), (b) it is allowed to use the variable reference [\\$xsl:original](#), and (c) the fact that it overrides C_Q affects the way that references from other components are rebound.
3. Component T_P is a new component declared locally in P .

The general rules for [reference binding](#) can now be stated:

1. If the [containing package](#) of a component C_P is P , then all [symbolic references](#) in C_P are bound to components whose [containing package](#) is P .
2. When a package P uses a package Q , then for every component C_Q in Q , there is a **corresponding component** C_P in P , as described in [3.5.3.2 Accepting Components](#).
3. Given a component C_P whose [containing package](#) and [declaring package](#) are the same package P , then (as a consequence of rules elsewhere in this specification) for every [symbolic reference](#) D within C_P , other than a reference using the name `xsl:original`, there will always be exactly one non-hidden component D_P whose containing package is P and whose [symbolic identifier](#) matches D (otherwise a static error will have been reported). The reference is then bound to D_P .
4. In the case of a component reference using the name `xsl:original`, this will in general appear within a component C_P that overrides a component C_Q whose corresponding component in P is C_{PQ} , and the `xsl:original` reference is bound to C_{PQ} .
5. Given a component C_P whose [containing package](#) P is a different package from its [declaring package](#) R (that is, C_P is present in P by virtue of an [xsl:use-package](#) declaration referencing package Q , which may or may not be the same as R), then the component bindings in C_P are derived from the component bindings in the corresponding component C_Q as follows: if the component binding within C_Q is to a component D_Q , then:
 - a. If D_Q is overridden within P by a component D_P , then the reference is bound to D_P ;
 - b. Otherwise, the reference is bound to the component D_{PQ} in P whose corresponding component in Q is D_Q .

When reference resolution is performed on a package that is intended to be used as a [stylesheet](#) (that is, for the [top-level package](#)), there must be no symbolic references referring to components whose visibility is [abstract](#) (that is, an implementation must be provided for every abstract component).

[ERR XTSE3080] It is a [static error](#) if a [top-level package](#) (as distinct from a [library package](#)) contains components whose visibility is [abstract](#).

Note:

This means that abstract components must either be overridden in a using package by a component that supplies a real implementation, or they must be accepted with `visibility="hidden"` (see [3.5.3.2 Accepting Components](#)), which has the effect that any invocation of the component raises a [dynamic error](#).

Note:

Unresolved references are allowed at the module level but not at the package level. A stylesheet module can contain references to components that are satisfied only when the module is imported into another module that declares the missing component.

Note:

The process of resolving references (or linking) is critical to an implementation that uses separate compilation. One of the aims of these rules is to ensure that when compiling a package, it is always possible to determine the signature of called functions, templates, and other components. A further aim is to establish unambiguously in what circumstances components can be overridden, so that compilers know when it is possible to perform optimizations such as inlining of function and variable references.

Suppose a public template T calls a private function F . When the package containing these two components is referenced by a using package, the template remains public, while the function becomes hidden. Because the function becomes hidden, it can no longer conflict with any other function of the same name, or be overridden by any other function; at this stage the compiler knows exactly which function T will be calling, and can perform optimizations based on this knowledge.

The mechanism for resolving component references described in this section is consistent with the mechanism used for binding function and variable references described in the XPath specification. XPath requires these variable and function names to be present in the static context for an XPath expression. XSLT ensures that all the non-hidden functions, global variables, and global parameters in a package are present in the static context for every XPath expression that appears in that package, along with required information such as the type of a variable and the signature of a function.

Example: Named Component References in Inline Functions

Named component references within inline functions follow the standard rules, but the rules need to be interpreted with care. Suppose that in package *P* we find the declarations:

```
<xsl:variable name="v" as="xs:integer" visibility="public" select="3"/>

<xsl:function name="f:factory" as="function(*)" visibility="final">
  <xsl:sequence select="function() {$v}" />
</xsl:function>
```

and that in a using package *Q* we find:

```
<xsl:use-package name="P">
  <xsl:override>
    <xsl:variable name="v" as="xs:integer" select="4"/>
  </xsl:override>
</xsl:use-package>

<xsl:template name="xsl:initial-template">
  <v value="{f:factory()()}" />
</xsl:template>
```

The correct output here is `<v value="4" />`.

The explanation for this is as follows. Package *Q* contains a function $f:\text{factory}_{QP}$ whose declaring package is *P* and whose containing package is *Q*. The symbolic reference $\$v$ within the body of this function is resolved in the normal way; since the containing package is *Q*, it is resolved to the global variable v_Q : that is, the overriding declaration of $\$v$ that appears within the `xsl:override` element within package *Q*, whose value is 4.

In terms of internal implementation, one way of looking at this is that the anonymous function returned by `f:factory` contains within its closure bindings for the global variables and functions that the anonymous function references; these bindings are inherited from the component bindings of the component that lexically contains these symbolic references, which in this case is `f:factory`, and more specifically the version of the `f:factory` component in package *Q*.

3.5.3.6 [Dynamic References to Components](#)

There are several functions in which a dynamically-evaluated QName is used to identify a component: these include `key`, `accumulator-before`, `accumulator-after`, `function-lookup`^{F030}, and `function-available`. Dynamic references can also occur in the XPath expression supplied to the `xsl:evaluate` instruction.

In all these cases, the set of components that are available to be referenced are those that are declared in the package where this function call appears, including components declared within an `xsl:override` declaration in that package, but excluding components declared with `visibility="abstract"`. If the relevant component has been overridden in a different package, the overriding declarations are not considered.

If one of these functions (for example [key](#) or [accumulator-before](#)) is invoked via a dynamic function invocation, then the relevant package is the one in which the function item is created (using a construct such as `key#2`, `key('my-key', ?)`, or `function-lookup($KEYFN, 2)`). Function items referring to context-dependent functions bind the context at the point where the function item is created, not the context at the point where the function item is invoked.

Note:

This means that if a package wishes to make a key available for use by a calling package, it can do so by creating a public global variable whose value is a partial application of the [key](#) function:

```
<xsl:variable name="get-order" select="key('orders-key', ?, ?)" />
```

which the calling code can invoke as `$get-order('123-456', /)`.

3.5.4 Overriding Template Rules from a Used Package

The rules in the previous section apply to named components including functions, named templates, global variables, and named attribute sets. The rules for [modes](#), and the [template rules](#) appearing within a mode, are slightly different.

The unnamed mode is local to a package: in effect, each package has its own private unnamed mode, and the unnamed mode of one package does not interact with the unnamed mode of any other package. An [xsl:apply-templates](#) instruction with no `mode` attribute is treated as a [symbolic reference](#) to the default mode defined for that instruction (see [3.7.2 The default-mode Attribute](#)), which in turn defaults to the [unnamed mode](#). Because the unnamed mode always has private visibility, it cannot be overridden in another package.

A named mode may be declared in an [xsl:mode](#) declaration as being either `public`, `private`, or `final`. The values of the `visibility` attribute are interpreted as follows:

Visibility Values for Named Modes, and their Meaning

Value	Meaning
<code>public</code>	A using package may use xsl:apply-templates to invoke templates in this mode; it may also declare additional template rules in this mode, which are selected in preference to template rules in the used package. These may appear only as children of the xsl:override element within the xsl:use-package element.
<code>private</code>	A using package may neither reference the mode nor provide additional templates in this mode; the name of the mode is not even visible in the using package, so no such attempt is possible. The using package can use the same name for its own modes without risk of conflict.
<code>final</code>	A using package may use xsl:apply-templates to invoke templates in this mode, but it must not provide additional template rules in this mode.

As with other named components, an [`xsl:use-package`](#) declaration may contain an [`xsl:accept`](#) element to control the visibility of a mode acquired from the **used** package. The allowed values of its `visibility` attribute are **public**, **private**, and **final**.

The [`xsl:mode`](#) declaration itself must not be overridden. A using package must not contain an [`xsl:mode`](#) declaration whose name matches that of a **public** or **final** [`xsl:mode`](#) component accepted from a used package.

The [`xsl:expose`](#) and [`xsl:accept`](#) elements may be used to reduce the visibility of a mode in a using package; the same rules apply in general, though some of the rules are not applicable because, for example, modes cannot be **abstract**.

It is not possible for a package to combine the template rules from two other packages into a single mode. When [`xsl:apply-templates`](#) is used without specifying a mode, the chosen template rules will always come from the same package; when it is used with a named mode, then they will come from the package where the mode is defined, or any package that uses that package and adds template rules to the mode. If two template rules defined in different packages match the same node, then the rule in the using package wins over any rule in the used package; this decision is made before taking other factors such as import precedence and priority into account.

A static error occurs if two modes with the same name are visible within a package, either because they are both declared within the package, or because one is declared within the package and the other is acquired from a used package, or because both are accepted from different used packages.

The rules for matching template rules by [`import precedence`](#) and [`priority`](#) operate as normal, with the addition that template rules declared within an [`xsl:use-package`](#) element have higher precedence than any template rule declared in the used package. More specifically, given an [`xsl:apply-templates`](#) instruction in package *P*, naming a mode *M* that is declared in a used package *Q* and is overridden in *P*, the search order for template rules is:

1. Rules declared within *P* (specifically, [`xsl:template`](#) rules declared as children of an [`xsl:override`](#) element within the [`xsl:use-package`](#) element that references package *Q*). If there are multiple rules declared within *P* that match a selected node, they are resolved on the basis of their explicit or implicit [`priority`](#), and if the priorities are equal, the last one in [`declaration order`](#) wins.
2. Rules declared within *Q*, taking [`import precedence`](#), [`priority`](#), and [`declaration order`](#) into account in the usual way (see [6.4 Conflict Resolution for Template Rules](#)).
3. Built-in template rules (see [6.7 Built-in Template Rules](#)) selected according to the `on-no-match` attribute of the [`xsl:mode`](#) declaration (in *Q*), or its default.

If the mode is overridden again in a package *R* that uses *P*, then this search order is extended by adding *R* at the start of the search list, and so on recursively.

Note:

If existing XSLT code has been written to use template rules in the unnamed mode, a convenient way to incorporate this code into a [`library package`](#) is to add a stub module that defines a new named **public** or **final** mode, in which there is a single template rule whose content is the single instruction `<xsl:apply-templates select=". "/>`. This in effect redirects [`xsl:apply-templates`](#) instructions using the named mode to the rules defined in the unnamed mode.

[3.5.4.1 Requiring Explicit Mode Declarations](#)

In previous versions of XSLT, modes were implicitly declared by simply using a mode name in the `mode` attribute of [xsl:template](#) or [xsl:apply-templates](#). XSLT 3.0 introduces the ability to declare a mode explicitly using an [xsl:mode](#) declaration (see [6.6.1 Declaring Modes](#)).

By default, within a package that is defined using an explicit [xsl:package](#) element, all modes must be explicitly declared. In an implicit package, however (that is, one rooted at an [xsl:stylesheet](#) or [xsl:transform](#) element), modes can be implicitly declared as in previous XSLT versions.

The `declared-modes` attribute of [xsl:package](#) determines whether or not modes that are referenced within the package must be explicitly declared. If the value is `yes` (the default), then it is an error to use a mode name unless the package either contains an explicit [xsl:mode](#) declaration for that mode, or accepts the mode from a used package. If the value is `no`, then this is not an error.

This attribute affects all modules making up the package, it is not confined to declarations appearing as children of the [xsl:package](#) element.

[ERR XTSE3085] It is a [static error](#), when the effective value of the `declared-modes` attribute of an [xsl:package](#) element is `yes`, if the package contains an explicit reference to an undeclared mode, or if it implicitly uses the unnamed mode and the unnamed mode is undeclared.

For the purposes of the above rule:

1. A mode is **declared** if either of the following conditions is true:
 - a. The package contains an [xsl:mode](#) declaration for that mode.
 - b. The mode is a public or final mode accepted from a used package.
2. The offending reference may be either an explicit mode name, or the token `#unnamed` treated as a reference to the unnamed mode, or a defaulted mode attribute, and it may occur in any of the following:
 - a. The `mode` attribute of an [xsl:template](#) declaration
 - b. The `mode` attribute of an [xsl:apply-templates](#) instruction
 - c. An `[xsl:]default-mode` attribute.
3. A package **implicitly uses the unnamed mode** if either of the following conditions is true:
 - a. There is an [xsl:apply-templates](#) element with no `mode` attribute, and with no ancestor-or-self having an `[xsl:]default-mode` attribute.
 - b. There is an [xsl:template](#) element with a `match` attribute and no `mode` attribute, and with no ancestor-or-self having an `[xsl:]default-mode` attribute.

[3.5.5 Declarations Local to a Package](#)

The [xsl:import](#) and [xsl:include](#) declarations are local to a package.

Declarations of [keys](#), [accumulators](#), [decimal formats](#), namespace aliases (see [11.1.4 Namespace Aliasing](#)), [output definitions](#), and [character maps](#) within a package have local scope within that package — they are all effectively private. The elements that declare these constructs do not have a `visibility` attribute. The unnamed decimal format and the unnamed output format are also local to a package.

If [xsl:strip-space](#) or [xsl:preserve-space](#) declarations appear within a [library package](#), they only affect calls to the [doc^{FO30}](#) or [document](#) functions appearing within that package. Such a declaration within the [top-level](#)

[package](#) additionally affects stripping of whitespace in the document that contains the [global context item](#).

An [xsl:decimal-format](#) declaration within a package applies only to calls on [format-number](#)^{FO30} appearing within that package.

An [xsl:namespace-alias](#) declaration within a package applies only to literal result elements within the same package.

An [xsl:import-schema](#) declaration within a package adds the names of the imported schema components to the static context for that package only; these names are effectively private, in the sense that they do not become available for use in any other packages. However, the names of schema components must be consistent across the stylesheet as a whole: it is not possible for two different packages within a stylesheet to use a type-name such as `part-number` to refer to different schema-defined simple or complex types.

Type names used in the interface of public components in a package (for example, in the arguments of a function) must be respected by callers of those components, in the sense that the caller must supply values of the correct type. Often this will mean that the using component, if it contains calls on such interfaces, must itself import the necessary schema components. However, the requirement for an explicit schema import applies only where the package contains explicit use of the names of schema components required to call such interfaces.

Note:

For example, suppose a [library package](#) contains a function which requires an argument of type `mfg:part-number`. The caller of this function must supply an argument of the correct type, but does not need to import the schema unless it explicitly uses the schema type name `mfg:part-number`. If it obtains an instance of this type from outside the package, for example as the result of another function call, then it can supply this instance to the acquired function even though it has not imported a schema that defines this type.

At execution time, the schema available for validating instance documents contains (at least) the union of the schema components imported into all constituent packages of the stylesheet.

3.5.6 Declaring the Global Context Item

The [xsl:global-context-item](#) element is used to declare whether a [global context item](#) is required, and if so, what its [required type](#) is.

The element is a [declaration](#) that can appear at most once in any stylesheet module; and if more than one [xsl:global-context-item](#) declaration appears within a [package](#), then the declarations must be consistent. Specifically, all the attributes **MUST** have semantically-equivalent values.

Note:

This means that omitting an attribute is equivalent to specifying its default value explicitly; and purely lexical variations, such as the presence of whitespace in an attribute value, are not considered significant.

[ERR XTSE3087] It is a [static error](#) if more than one [xsl:global-context-item](#) declaration appears within a [stylesheet module](#), or if several modules within a single [package](#) contain inconsistent [xsl:global-context-item](#).

item declarations

If there is no [xsl:global-context-item](#) declaration for a package, this is equivalent to specifying the empty element <xsl:global-context-item/>, which imposes no constraints.

```
<!-- Category: declaration -->
<xsl:global-context-item
  as? = item-type
  use? = "required" | "optional" | "absent" />
```

The use attribute takes the value required, optional, or absent. The default is optional.

- If the value required is specified, then there must be a global context item.
- If the value optional is specified, or if the attribute is omitted, or if the [xsl:global-context-item](#) element is omitted, then there may or may not be a global context item.
- If the value absent is specified, then the global focus (context item, position, and size) will be [absent](#)

Note:

This specification does not define whether supplying a global context item in this situation results in an error or warning, or whether the supplied context item is simply ignored.

If the as attribute is present then its value must be an [ItemType^{XP30}](#). If the attribute is omitted this is equivalent to specifying as="item()".

The as attribute defines the required type of the global context item. The default value is as="item()". If a global context item is supplied then it must conform to the required type, after conversion (if necessary) using the [function conversion rules](#).

[ERR XTSE3089] It is a [static error](#) if the as attribute is present when use="absent" is specified.

The global context item is available only within the [top-level package](#). If a valid [xsl:global-context-item](#) declaration appears within a [library package](#), then it is ignored, unless it specifies use="required", in which case an error is signaled: [see [ERR XTTE0590](#)].

Note:

In earlier releases of this specification, the [global context item](#) and the [initial match selection](#) were essentially the same thing, often referred to as the *principal source document*. In XSLT 3.0, they have been separated: the global context item is a single item accessible to the initializers of global variables as the value of the expression . (dot), while the initial match selection is a sequence of nodes or other items supplied to an initial implicit [xsl:apply-templates](#) invocation.

APIs that were originally designed for use with earlier versions of XSLT are likely to bundle the two concepts together.

With a streamable processor, the [initial match selection](#) can consist of streamed nodes, but the [global context item](#) is always [grounded](#), because it is available to all global variables and there is no control over the sequence of processing.

A [type error](#) is signaled if there is a [package](#) with an [`xsl:global-context-item`](#) declaration specifying a required type that does not match the supplied [global context item](#). The error code is the same as for [`xsl:param`](#): [see [ERR_XTTE0590](#)].

Note:

If the `ItemType` is one that can only be satisfied by a schema-validated input document, for example `as="schema-element(invoice)"`, the [processor](#) may interpret this as a request to apply schema validation to the input. Similarly, if the `KindTest` indicates that an element node is required, the processor may interpret this as a request to supply the document element rather than the document node of a supplied input document.

3.5.7 [Worked Example of a Library Package](#)

The example in this section illustrates the use of overrides to customize or extend a (fictional) library package named `http://example.com/csv-parser`, which provides a parsing function for data formatted as lines containing comma-separated values. For simplicity of exposition, the example shows a simple, naive implementation; a realistic CSV parser would be more complicated and make the example harder to follow.

3.5.7.1 [Default Functionality of the CSV Package](#)

The basic functionality of the package is provided by the function `csv:parse`, which expects a string parameter named `input`. By default, the function parses the input into lines, and breaks lines on commas, returning as result an element named `csv` containing one `row` element per line, each `row` containing a sequence of `field` elements.

A simple stylesheet which uses this library and applies it to a string might look like the following. The initial template applies `csv:parse` to a suitable string and returns a copy of the result:

```

<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:csv="http://example.com/csv"
  exclude-result-prefixes="xs csv"
  version="3.0">

  <xsl:output indent="yes"/>

  <xsl:use-package name="http://example.com/csv-parser"
    package-version="*"/>

  <!-- example input "file" -->
  <xsl:variable name="input" as="xs:string">
    name,id,postal code
    "Abel Braaksma",34291,1210 KA
    "Anders Berglund",473892,9843 ZD
  </xsl:variable>

  <!-- entry point -->
  <xsl:template name="xsl:initial-template">
    <xsl:copy-of select="csv:parse($input)"/>
  </xsl:template>

</xsl:stylesheet>

```

The result returned by this stylesheet would be:

```

<csv>
  <row>
    <field quoted="no">name</field>
    <field quoted="no">id</field>
    <field quoted="no">postal code</field>
  </row>
  <row>
    <field quoted="yes">Abel Braaksma</field>
    <field quoted="no">34291</field>
    <field quoted="no">1210 KA</field>
  </row>
  <row>
    <field quoted="yes">Anders Berglund</field>
    <field quoted="no">473892</field>
    <field quoted="no">9843 ZD</field>
  </row>
</csv>

```

Variations on this default behavior are achieved by overriding selected declarations in the package, as described below.

3.5.7.2 *Package Structure*

The package module itself is version 1.0.0 of a package called `http://example.com/csv-parser`; it has the following structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:package
  name="http://example.com/csv-parser"
  package-version="1.0.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:csv="http://example.com/csv"
  exclude-result-prefixes="xs csv"
  declared-modes="yes"
  version="3.0">

  <!--* Mode declarations ... *-->
  <!--* Variable declarations ... *-->
  <!--* Attribute-set declaration ... *-->
  <!--* Function declarations ... *-->
  <!--* Templates ... *-->

</xsl:package>
```

The contents of the package (represented here by comments) are described more fully below.

3.5.7.3 *[The csv:parse Function and its User-customization Hooks](#)*

The `csv:parse` function is final and cannot be overridden. As can be seen from the code below, it (1) parses its `input` parameter into lines, (2) calls function `csv:preprocess-line` on each line, then (3) applies the templates of mode `csv:parse-line` to the pre-processed value. The result is then (4) processed again by mode `csv:post-process`.

```
<xsl:function name="csv:parse" visibility="final">
  <xsl:param name="input" as="xs:string"/>
  <xsl:variable name="result" as="element()">
    <csv>
      <xsl:apply-templates
        select="(tokenize($input, $csv:line-separator)
          ! csv:preprocess-line(.))"
        mode="csv:parse-line"/>
    </csv>
  </xsl:variable>
  <xsl:apply-templates select="$result"
    mode="csv:post-process"/>
</xsl:function>
```

The default code for this processing is given below. Each part of the processing except the first (the tokenization into lines) can be overridden by the user of the package.

3.5.7.4 *[Breaking the Input into Lines](#)*

The first user-customization hook is given by the global variable `csv:line-separator`, which specifies the line separator used to break the input string into lines. It can be overridden by the user if need be. The default declaration attempts to handle the line-separator sequences used by most common operating systems in text files:

```
<xsl:variable name="csv:line-separator"
    as="xs:string"
    select="'\r\n?|\n\r?'"
    visibility="public"/>
```

3.5.7.5 Pre-processing the Lines

The function `csv:preprocess-line` calls `normalize-space()` on its argument:

```
<xsl:function name="csv:preprocess-line"
    as="xs:string?"
    visibility="public">
    <xsl:param name="line" as="xs:string"/>
    <xsl:sequence select="normalize-space($line)" />
</xsl:function>
```

Because the function is declared `public`, it can be overridden by a user. (This might be necessary, for example, if whitespace within quoted strings needs to be preserved.)

3.5.7.6 The Mode `csv:parse-line`

By default, the mode `csv:parse-line` parses the current item (this will be one line of the input data) into fields, using mode `csv:parse-field` on the individual fields and (by default) wrapping the result in a `row` element.

The mode is declared with `visibility="public"` to allow it to be called from elsewhere and overridden:

```
<xsl:mode name="csv:parse-line" visibility="public"/>

<xsl:template match=". " mode="csv:parse-line">
    <row>
        <xsl:apply-templates
            select="tokenize(., $csv:field-separator)"
            mode="csv:parse-field"/>
    </row>
</xsl:template>
```

This relies on the variable `csv:field-separator`, which is a comma by default but which can be overridden by the user to parse tab-separated data or data with other delimiters.

```
<xsl:variable name="csv:field-separator"
    as="xs:string"
    select="','"
    visibility="public"/>
```

The default implementation of `csv:parse-line` does not handle occurrences of the field separator occurring within quoted strings. The user can add templates to the mode to provide that functionality.

3.5.7.7 Mode `csv:parse-field`

Mode `csv:parse-field` processes the current item as a field; by default it strips quotation marks from the value, calls the function `csv:preprocess-field()` on it, and wraps the result in a `field` element, which carries the attributes declared in the attribute set `csv:field-attributes`.

```
<xsl:template match="."
    mode="csv:parse-field"
    expand-text="yes">
    <xsl:variable name="string-body-pattern"
        as="xs:string"
        select="'^' || $csv:validated-quote || ']*'"/>
    <xsl:variable name="quoted-value"
        as="xs:string"
        select="$csv:validated-quote
            || $string-body-pattern
            || $csv:validated-quote"/>
    <xsl:variable name="unquoted-value"
        as="xs:string"
        select="'.+'"/>

    <field xsl:use-attribute-sets="csv:field-attributes">{
        csv:preprocess-field(
            replace(.,
                $quoted-value || '||' || $unquoted-value,
                '$1$2'))
    }</field>
</xsl:template>
```

The attribute set `csv:field-attributes` includes, by default, a `quoted` attribute which has the values `yes` or `no` to show whether the input value was quoted or not.

```
<xsl:attribute-set name="csv:field-attributes"
    visibility="public">
    <xsl:attribute name="quoted"
        select="if (starts-with(., $csv:validated-quote))
            then 'yes'
            else 'no'"/>
</xsl:attribute-set>
```

The mode `csv:parse-field` is declared with `visibility="public"` to allow it to be called from elsewhere and overridden; it specifies `on-no-match="shallow-copy"` so that any string not matching a template will simply be copied:

```
<xsl:mode name="csv:parse-field"
    on-no-match="shallow-copy"
    visibility="public"/>
```

3.5.7.8 The `csv:quote` Variable

The variable `csv:quote` can be used to specify the character used in a particular input stream to quote values.

```
<xsl:variable name="csv:quote"
    as="xs:string"
    select="'"'" visibility="public"/>
```

The template given above assumes that the variable is one character long. To ensure that any overriding value of the variable is properly checked, references to the value use a second variable `csv:validated-quote`, which is declared `private` to ensure that the checking cannot be disabled.

```
<xsl:variable name="csv:validated-quote" visibility="private"
    as="xs:string" select="
        if (string-length($csv:quote) ne 1)
        then error(xs:QName('csv:ERR001'),
                  'Incorrect length for $csv:quote, should be 1')
        else $csv:quote"/>
```

When the value of `csv:quote` is not exactly one character long, the reference to `csv:validated-quote` will cause an error (`csv:ERR001`) to be raised.

3.5.7.9 [The `csv:preprocess-field` Function](#)

The function `csv:preprocess-field` is called on each field after any quotation marks are stripped and before it is written out as the value of a `field` element:

```
<xsl:function name="csv:preprocess-field"
    as="xs:string">
    <xsl:param name="field"
        as="xs:string"/>
    <xsl:sequence select="$field"/>
</xsl:function>
```

As can be seen, the function does nothing but return its input; its only purpose is to provide the opportunity for the user to supply a suitable function to be invoked at this point in the processing of each field.

3.5.7.10 [The Mode `csv:post-process`](#)

The mode `csv:post-process` is intended solely as a hook for user code. By default, it does nothing.

The package defines no templates for this mode; the mode definition makes it return a copy of its input:

```
<xsl:mode name="csv:post-process"
    on-no-match="shallow-copy"
    visibility="public"/>
```

3.5.7.11 [Overriding the Default Behavior](#)

As can be seen from the code shown above, the package provides several opportunities for users to override the default behavior:

- The global variables `csv:line-separator`, `csv:field-separator`, and `csv:quote` can be overridden to specify the character strings used to separate lines and fields and to quote individual field values.
- The function `csv:preprocess-line` can be overridden to do more (or less) than stripping white space; the function `csv:preprocess-field` can be overridden to process individual field values.
- Templates can be added to the modes `csv:parse-line`, `csv:parse-field`, and `csv:post-process` to change their behavior.
- The attribute set `csv:field-attributes` can be overridden to specify a different set of attributes (or none) for `field` elements.

The following XSLT stylesheet illustrates the use of the `xsl:override` element to take advantage of several of these opportunities:

```

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:csv="http://example.com/csv"
  exclude-result-prefixes="xs csv"
  version="3.0">

  <xsl:output indent="yes"/>

  <xsl:use-package name="http://example.com/csv-parser"
    package-version="*"
    <xsl:override>
      <!-- Change the root element from 'csv' to 'root' -->
      <xsl:template match="csv" mode="csv:post-process">
        <root>
          <xsl:apply-templates mode="csv:post-process"/>
        </root>
      </xsl:template>

      <!-- add an extra attribute that uses the context item -->
      <xsl:attribute-set name="csv:field-attributes"
        use-attribute-sets="xsl:original">
        <xsl:attribute name="type"
          select="if (. castable as xs:decimal)
            then 'numeric'
            else 'string'"/>
      </xsl:attribute-set>

      <!-- use semicolon not comma between fields -->
      <xsl:variable name="csv:field-separator"
        as="xs:string" select="';'"
        visibility="public"/>

      <!-- prevent empty rows from appearing with empty lines -->
      <xsl:function name="csv:preprocess-line"
        as="xs:string?"
        visibility="public">
        <xsl:param name="line" as="xs:string"/>
        <xsl:variable name="norm-line"
          select="normalize-space(xsl:original($line))"/>
        <xsl:sequence select="if (string-length($norm-line) > 0)
          then $norm-line
          else ()"/>
      </xsl:function>
    </xsl:override>
  </xsl:use-package>

  <!-- example input "file" -->
  <xsl:variable name="input" as="xs:string">
    name;id;postal code
    "Braaksma Abel";34291;1210 KA
    "Berglund Anders";473892;9843 ZD
  </xsl:variable>

  <!-- entry point -->

```

```

<xsl:template name="xsl:initial-template">
  <xsl:copy-of select="csv:parse($input)" />
</xsl:template>

</xsl:stylesheet>

```

Note:

- As it does elsewhere, the visibility of components declared within `xsl:override` defaults to `private`; to keep the component public, it is necessary to specify visibility explicitly.
- The types and optionality of all function parameters must match those of the function being overridden; for function overriding to be feasible, packages must document the function signature thoroughly.
- The names, types, and optionality of all named-template parameters must match those of the template being overridden; for overriding to be feasible, packages must document the template signature thoroughly.
- The values for the attributes in the attribute set `csv:field-attributes` are calculated once for each element for which the attribute set is supplied; the `select` attributes which determine the values can thus refer to the context item. Here, the value specification for the `type` attribute checks to see whether the string value of the context item is numeric by inquiring whether it can be cast to decimal, and sets the value for the `type` attribute accordingly.

The result returned by this stylesheet would be:

```

<root>
  <row>
    <field quoted="no" type="string">name</field>
    <field quoted="no" type="string">id</field>
    <field quoted="no" type="string">postal code</field>
  </row>
  <row>
    <field quoted="yes" type="string">Braaksma Abel</field>
    <field quoted="no" type="numeric">34291</field>
    <field quoted="no" type="string">1210 KA</field>
  </row>
  <row>
    <field quoted="yes" type="string">Berglund Anders</field>
    <field quoted="no" type="numeric">473892</field>
    <field quoted="no" type="string">9843 ZD</field>
  </row>
</root>

```

3.6 Stylesheet Modules

[**DEFINITION:** A `package` 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 [\[XDM 3.0\]](#)). 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.

The principal stylesheet module of a package may take one of three forms:

- A package manifest, as described in [3.5 Packages](#), which is a subtree rooted at an [`xsl:package`](#) element
- An implicit package, which is a subtree rooted at an [`xsl:stylesheet`](#) or [`xsl:transform`](#) element. This is transformed automatically to a package as described in [3.5 Packages](#).
- A simplified stylesheet, which is a subtree rooted at a literal result element, as described in [3.8 Simplified Stylesheet Modules](#). This is first converted to an implicit package by wrapping it in an [`xsl:stylesheet`](#) element using the transformation described in [3.8 Simplified Stylesheet Modules](#), and then to an explicit package (rooted at an [`xsl:package`](#) element) using the transformation described in [3.5 Packages](#).

A stylesheet module other than the principal stylesheet module of a package may take either of two forms:

- [DEFINITION: A **standard stylesheet module**, which is a subtree rooted at an [`xsl:stylesheet`](#) or [`xsl:transform`](#) element.]
- [DEFINITION: A **simplified stylesheet**, which is a subtree rooted at a [literal result element](#), as described in [3.8 Simplified Stylesheet Modules](#). This is first converted to a **standard stylesheet module** by wrapping it in an [`xsl:stylesheet`](#) element using the transformation described in [3.8 Simplified Stylesheet Modules](#).]

Whichever of the above forms a module takes, the outermost element ([`xsl:package`](#), [`xsl:stylesheet`](#), or a [literal result element](#)) MAY either be the outermost element of an XML document, or it MAY be a child of some (non-XSLT) element in a host document.

[DEFINITION: A stylesheet module whose outermost element is the child of a non-XSLT element in a host document is referred to as an **embedded stylesheet module**. See [3.12 Embedded Stylesheet Modules](#).]

[3.7 Stylesheet Element](#)

```

<xsl:stylesheet
    id? = id
    version = decimal
    default-mode? = eqname | "#unnamed"
    default-validation? = "preserve" | "strip"
    input-type-annotations? = "preserve" | "strip" | "unspecified"
    default-collation? = uris
    extension-element-prefixes? = prefixes
    exclude-result-prefixes? = prefixes
    expand-text? = boolean
    use-when? = expression
    xpath-default-namespace? = uri >
    <!-- Content: (declarations) -->
</xsl:stylesheet>
```

```

<xsl:transform
    id? = id
    version = decimal
    default-mode? = eqname | "#unnamed"
    default-validation? = "preserve" | "strip"
    input-type-annotations? = "preserve" | "strip" | "unspecified"
    default-collation? = uris
    extension-element-prefixes? = prefixes
    exclude-result-prefixes? = prefixes
    expand-text? = boolean
    use-when? = expression
    xpath-default-namespace? = uri >
    <!-- Content: (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](#).

The **version** attribute indicates the version of XSLT that the stylesheet module requires. The attribute is REQUIRED.

[ERR_XTSE0110] The value of the **version** attribute MUST be a number: specifically, it MUST be a valid instance of the type [xs:decimal](#) as defined in [\[XML Schema Part 2\]](#).

The **version** attribute is intended to indicate the version of the XSLT specification against which the stylesheet is written. In a stylesheet written to use XSLT 3.0, the value SHOULD normally be set to `3.0`. If the value is numerically less than `3.0`, the stylesheet is processed using the rules for [backwards compatible behavior](#) (see [3.9 Backwards Compatible Processing](#)). If the value is numerically greater than `3.0`, the stylesheet is processed using the rules for [forwards compatible behavior](#) (see [3.10 Forwards Compatible Processing](#)).

The effect of the **input-type-annotations** attribute is described in [4.4.1 Stripping Type Annotations from a Source Tree](#).

The `[xsl:]default-validation` attribute defines the default value of the `validation` attribute of all relevant instructions appearing within its scope. For details of the effect of this attribute, see [25.4 Validation](#).

[ERR XTSE0120] An `xsl:stylesheet`, `xsl:transform`, or `xsl:package` element MUST NOT have any text node children. (This rule applies after stripping of `whitespace text nodes` as described in [4.3 Stripping Whitespace from the Stylesheet](#).)

[DEFINITION: An element occurring as a child of an `xsl:package`, `xsl:stylesheet`, `xsl:transform`, or `xsl:override` 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 [3.7.3 User-defined Data Elements](#))].

The [declaration](#) elements permitted in the `xsl:stylesheet` element are:

```
xsl:accumulator
xsl:attribute-set
xsl:character-map
xsl:decimal-format
xsl:function
xsl:global-context-item
xsl:import
xsl:import-schema
xsl:include
xsl:key
xsl:mode
xsl:namespace-alias
xsl:output
xsl:param
xsl:preserve-space
xsl:strip-space
xsl:template
xsl:use-package
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.

The child elements of the `xsl:stylesheet` element may appear in any order. In most cases, the ordering of these elements does not affect the results of the transformation; however:

- As described in [6.4 Conflict Resolution for Template Rules](#), when two template rules with the same [priority](#) match the same nodes, there are situations where the order of the template rules will affect which is chosen.
- Forwards references to [static variables](#) are not allowed in [static expressions](#).

3.7.1 [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, when it appears on an element E , is used to specify the default collation used by all XPath expressions appearing in attributes or [text value templates](#) that have E as an ancestor, 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 reference, 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 include as the last collation URI in the list either the Unicode Codepoint Collation or a collation in the UCA family (see [13.4 The Unicode Collation Algorithm](#)) with the parameter `fallback=yes`.

The `[xsl:]default-collation` attribute does not affect the collation used by [xsl:sort](#) or by [xsl:merge](#).

In the absence of an `[xsl:]default-collation` attribute, the default collation **MAY** be set by the calling application in an [implementation-defined](#) way. The recommended default, unless the user chooses otherwise, is to use the Unicode codepoint collation.

3.7.2 The `default-mode` Attribute

The `[xsl:]default-mode` attribute defines the default value for the `mode` attribute of all [xsl:template](#) and [xsl:apply-templates](#) elements within its scope.

More specifically, when an element E matches the pattern `(xsl:template[@match] | xsl:apply-templates)[not(@mode) or normalize-space(@mode) eq "#default"]` (using the Unicode codepoint collation), then the effective value of the `mode` attribute is taken from the value of the `[xsl:]default-mode` attribute of the innermost ancestor-or-self element of E that has such an attribute. If there is no such element, then the default is the [unnamed mode](#). This is equivalent to specifying `#unnamed`.

In addition, when the attribute appears on the [xsl:package](#), [xsl:stylesheet](#), or [xsl:transform](#) element of the [principal stylesheet module](#) of the [top-level package](#), it provides a default value for the [initial mode](#) used on stylesheet invocation.

The value of the `[xsl:]default-mode` attribute **MUST** either be an [EQName](#), or the token `#unnamed` which refers to the [unnamed mode](#).

Note:

This attribute is provided to support an approach to stylesheet modularity in which all the template rules for one [mode](#) are collected together into a single [stylesheet module](#). Using this attribute reduces the risk of forgetting to specify the mode in one or more places where it is needed, and it also makes it easier to reuse an existing stylesheet module that does not use modes in an application where modes are needed to avoid conflicts with existing template rules.

It is not necessary for the referenced mode to be explicitly declared in an [xsl:mode](#) declaration, unless this is mandated by the [declared-modes](#) attribute (which defaults to `yes` on an [xsl:package](#) element).

[3.7.3 User-defined Data Elements](#)

[**DEFINITION:** In addition to [declarations](#), the [xsl:stylesheet](#) element may contain among its children 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 an [xsl:stylesheet](#), [xsl:transform](#), or [xsl:package](#) 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 [3.2 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 [24 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.

[3.8 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="3.0"
      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="3.0"
                 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="3.0"
    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" select="/" />
                <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 `3.0` ; the value **MUST** be a valid instance of the type `xs:decimal` as defined in [\[XML Schema Part 2\]](#).

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 [template rules](#), [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 match selection](#), to be matched by the implicit `match="/"` template rule using the [unnamed mode](#).

3.9 Backwards Compatible Processing

[**DEFINITION:** The **effective version** of an element in a [stylesheet module](#) or [package manifest](#) is the decimal value of the `[xsl:]version` attribute (see [3.4 Standard Attributes](#)) on that element or on the innermost ancestor element that has such an attribute, excluding the `version` attribute on an [xsl:output](#) element.]

[**DEFINITION:** An element is processed with **backwards compatible behavior** if its [effective version](#) is less than `3.0`.]

Specifically:

- If the [effective version](#) is equal to `1.0`, then the element is processed with XSLT 1.0 behavior as described in [3.9.1 XSLT 1.0 Compatibility Mode](#).
- If the [effective version](#) is equal to `2.0`, then the element is processed with XSLT 2.0 behavior as described in [3.9.2 XSLT 2.0 Compatibility Mode](#).
- If the [effective version](#) is any other value less than `3.0`, the **RECOMMENDED** action is to report a static error; however, processors **MAY** recognize such values and process the element in an [implementation-defined](#) way.

Note:

XSLT 1.0 allowed the `version` attribute to take any decimal value, and invoked forwards compatible processing for any value other than 1.0. XSLT 2.0 allowed the attribute to take any decimal value, and invoked backwards compatible (i.e. 1.0-compatible) processing for any value less than 2.0. Some stylesheets may therefore be encountered that use values other than 1.0 or 2.0. In particular, the value 1.1 is sometimes encountered, as it was used at one stage in a draft language proposal.

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.

It is implementation-defined whether a particular XSLT 3.0 implementation supports backwards compatible behavior for any XSLT version earlier than XSLT 3.0.

[ERR XTDE0160] It is a dynamic error if an element has an effective version of V (with $V < 3.0$) when the implementation does not support backwards compatible behavior for XSLT version V .

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 3.0 processor are identical to the effects of processing the same stylesheet using a processor for an earlier version of XSLT. 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 3.0 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 3.0 and XPath 3.0 specifications, it is not defined by reference to earlier versions of the XSLT and XPath specifications.

To write a stylesheet that makes use of features that are new in version N , while also working with a processor that only supports XSLT version M ($M < N$), it is necessary to understand both the rules for backwards compatible behavior in XSLT version N , and the rules for forwards compatible behavior in XSLT version M . If the `xsl:stylesheet` element specifies `version="2.0"` or `version="3.0"`, then an XSLT 1.0 processor will ignore XSLT 2.0 and XSLT 3.0 declarations that were not defined in XSLT 1.0, for example `xsl:function` and `xsl:import-schema`. If any new XSLT 3.0 instructions are used (for example `xsl:evaluate` or `xsl:source-document`), or if new XPath 3.0 features are used (for example, new functions, or let expressions), then the stylesheet must provide fallback behavior that relies only on facilities available in the earliest XSLT version supported. 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.

3.9.1 XSLT 1.0 Compatibility Mode

[DEFINITION: An element in the stylesheet is processed with **XSLT 1.0 behavior** if its effective version is equal to 1.0.]

In this mode, if any attribute contains an XPath [expression](#), 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](#)^{XP30}. Expressions contained in [text value templates](#) are always evaluated with [XPath 1.0 compatibility mode](#) set to `false`, since this construct was not available in XSLT 1.0.

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 [24.1 Extension Functions](#).

Note:

This might appear to contradict the specification of XPath 3.0, 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 XSLT 1.0 compatibility mode is enabled. This is described separately for each such construct.

Processing an [instruction](#) with XSLT 1.0 behavior is not compatible with streaming. More specifically, and notwithstanding anything stated in [19 Streamability](#), an instruction that is processed with XSLT 1.0 behavior is [roaming](#) and [free-ranging](#), which has the effect that any construct containing such an instruction is not [guaranteed-streamable](#).

3.9.2 XSLT 2.0 Compatibility Mode

[**DEFINITION:** An element is processed with **XSLT 2.0 behavior** if its [effective version](#) is equal to 2.0.]

In this specification, no differences are defined for XSLT 2.0 behavior. An XSLT 3.0 processor will therefore produce the same results whether the [effective version](#) of an element is set to 2.0 or 3.0.

Note:

An XSLT 2.0 processor, by contrast, will in some cases produce different results in the two cases. For example, if the stylesheet contains an [xsl:iterate](#) instruction with an [xsl:fallback](#) child, an XSLT 3.0 processor will process the [xsl:iterate](#) instruction regardless whether the effective version is 2.0 or 3.0, while an XSLT 2.0 processor will report a static error if the effective version is 2.0, and will take the fallback action if the effective version is 3.0.

3.10 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 3.0, while retaining the ability to execute the stylesheet with an XSLT 3.0 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 [3.13.1 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 3.0 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 is processed with **forwards compatible behavior** if its [effective version](#) is greater than 3.0.]

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.

When an element is processed with forwards compatible behavior:

- If the element is in the XSLT namespace and appears as a child of the [`xsl:stylesheet`](#) element, and XSLT 3.0 does not allow the element to appear as a child of the [`xsl:stylesheet`](#) element, then the element and its content **MUST** be ignored.
- If the element has an attribute that XSLT 3.0 does not allow the element to have, then the attribute **MUST** be ignored.
- If the element is in the XSLT namespace and appears as a child of an element whose content model requires a [sequence constructor](#), and XSLT 3.0 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 3.0 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 3.0 [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 3.0, then the stylesheet can use an [xsl:message](#) element with `terminate="yes"` (see [23.1 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>
```

Note:

The XSLT 1.0 and XSLT 2.0 specifications did not anticipate the introduction of the [xsl:package](#) element. An XSLT 1.0 or 2.0 processor encountering this element will report a static error, regardless of the `version` setting.

This problem can be circumvented by using the simplified package syntax (whereby an [xsl:stylesheet](#) element is implicitly treated as [xsl:package](#)), or by writing the stylesheet code in a separate module from the package manifest, and using the separate module as the version of the stylesheet that is presented to a 2.0 processor.

3.11 Combining Stylesheet Modules

XSLT provides two mechanisms to construct a [package](#) 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.

3.11.1 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 reference, 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](#) MUST be either a [standard stylesheet module](#) or a [simplified stylesheet](#). It MUST NOT be a [package manifest](#). If it is a simplified stylesheet module then it is transformed into the equivalent standard stylesheet module by applying the transformation described in [3.8 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.

Note:

It is appropriate to use this error code when the resource cannot be retrieved, or when the retrieved resource is not well formed XML. If the resource contains XML that can be parsed but that violates the rules for stylesheet modules, then a more specific error code may be more appropriate.

3.11.2 [Stylesheet Inclusion](#)

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

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 [3.11.1 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*.]

Note:

A stylesheet level thus groups the [declarations](#) in a [package](#) by [import precedence](#): two declarations within a package are in the same stylesheet level if and only if they have the same import precedence.

[**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'*.

3.11.3 [Stylesheet Import](#)

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

A stylesheet module may import another [stylesheet module](#) using an [`xsl:import` declaration](#). Importing a stylesheet module is the same as including it (see [3.11.2 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 [3.11.1 Locating Stylesheet Modules](#).

[ERR XTSE0190] An [`xsl:import`](#) element must be a [top-level](#) element.

Example: Using `xsl:import`

For example,

```
<xsl:stylesheet version="3.0"
    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>
```

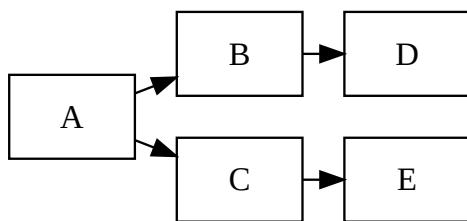
[**DEFINITION:** The stylesheet levels making up a stylesheet are treated as forming an **import tree**. In the import tree, each stylesheet level has one child for each `xsl:import` declaration that it contains.] The ordering of the children is the declaration order of the `xsl:import` declarations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

3.12 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](#)^{DM30}. This property will typically be set if the attribute is defined in a DTD as being of type ID, or if it 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 endorsed in respect of the media type `application/xml` by [\[RFC7303\]](#).

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 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="3.0"
      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 is the officially registered media type for XSLT: see [3.3 XSLT Media Type](#). However, browsers developed before this media type was registered are more likely to accept the unofficial designation `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](#).

3.13 Stylesheet Preprocessing

This specification provides two features that cause the raw stylesheet to be preprocessed as the first stage of static processing: elements may be conditionally included or excluded by means of an `[xsl:]use-when` attribute as described in [3.13.1 Conditional Element Inclusion](#), and attributes may be conditionally computed as described in [3.13.2 Shadow Attributes](#).

Note that many of the rules affecting the validity of stylesheet documents apply to a stylesheet after this preprocessing phase has been carried out.

3.13.1 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. 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 attribute is present and the [effective boolean value](#)^{XP30} 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`.

If the `xsl:package`, `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 [4.3 Stripping Whitespace from the Stylesheet](#).

The XPath expression used as the value of the `xsl:use-when` attribute follows the rules for [static expressions](#), including the rules for handling errors.

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'"/>
```

3.13.2 Shadow Attributes

When a no-namespace attribute name N is permitted to appear on an element in the [XSLT namespace](#) (provided that N does not start with an underscore), then a value V can be supplied for N in one of two ways:

- The conventional way is for an attribute node with name N and value V to appear in the XDM representation of the element node in the stylesheet tree.
- As an alternative, a shadow attribute may be supplied allowing the value V to be statically computed during the preprocessing phase. The shadow attribute has a name that is the same as the name N prefixed with an underscore, and the value of the shadow attribute is a [value template](#) in which all expressions enclosed between curly braces must be [static expressions](#). The value V is the result of evaluating the value template. If a shadow attribute is present, then any attribute node with name N (sharing the same parent element) is ignored.

For example, an `xsl:include` element might be written:

```
<xsl:include _href="common{$VERSION}.xsl"/>
```

allowing the stylesheet to include a specific version of a library module based on the value of a [static parameter](#).

Similarly, a [mode](#) might be declared like this:

```
<xsl:param name="streamable" as="xs:boolean"
           required="yes" static="yes"/>
<xsl:mode _streamable="{$streamable}" on-no-match="shallow-skip"/>
```

this allowing the streamability of the mode to be controlled using a [static parameter](#) (Note: this example relies on the fact that the `streamable` attribute accepts a boolean value, which means that the values `true` and `false` are accepted as synonyms of `yes` and `no`).

This mechanism applies to all attributes in the stylesheet where the attribute name is in no namespace and the name of the parent element is in the [XSLT namespace](#). This includes attributes that have static significance such as the `use-when` attribute, the `version` attribute, and the `static` attribute on [xsl:variable](#). The mechanism does not apply to shadow attributes (that is, it is not possible to invoke two stages of preprocessing by using two leading underscores). It does not apply to attributes of literal result elements, nor to attributes in a namespace such as the XML or XSLT namespace, nor to namespace declarations.

Note:

If a shadow attribute and its corresponding target attribute are both present in the stylesheet, the non-shadow attribute is ignored. This may be useful to make stylesheet code compatible across XSLT versions; an XSLT 2.0 processor operating in forwards compatible mode will ignore shadow attributes, and will require the target attribute to be valid.

Note:

The statement that the non-shadow attribute is ignored extends to error detection: it is not an error if the non-shadow attribute has an invalid value. However, this is not reflected in the schema for XSLT stylesheets, so validation using this schema may report errors in such cases.

Note:

An attribute whose name begins with an underscore is treated specially only when it appears on an element in the XSLT namespace. On a [literal result element](#), it is treated in the same way as any other attribute (that is, its effective value is copied to the result tree). On an [extension instruction](#) or [user-defined data element](#), as with other attributes on these elements, its meaning is entirely [implementation-defined](#).

Example: Using Shadow Attributes to Parameterize XPath Default Namespace

Although it is not usually considered good practice, it sometimes happens that variants or versions of an XML vocabulary exist in which the same local names are used, but in different namespaces. There is then a requirement to write code that will process source documents in a variety of different namespaces.

It is possible to define a static stylesheet parameter containing the target namespace, for example:

```
<xsl:param name="NS" as="xs:string" static="yes"
           select="'http://example.com/ns/one' />
```

And this can then be used to set the default namespace for XPath expressions:

```
_xpath-default-namespace="{$NS}"
```

However, it is not possible to put this shadow attribute on the [xsl:stylesheet](#) or [xsl:package](#) element of the principal stylesheet module, because at that point the variable \$NS is not in scope. A workaround is to create a stub stylesheet module which contains nothing but the static parameter declaration and an [xsl:include](#) of the stylesheet module containing the real logic. The static stylesheet parameter will then be in scope on the [xsl:stylesheet](#) element of the included stylesheet module, and the shadow attribute `_xpath-default-namespace="{$NS}"` can therefore appear on this [xsl:stylesheet](#) element.

Example: Using Shadow Attributes to Parameterize Selection of Elements

The following stylesheet produces a report giving information about selected employees. The predicate defining which employees are to be included in the report is supplied (as a string containing an XPath expression) in a static stylesheet parameter:

```

<xsl:param name="filter" static="yes"
           as="xs:string" select="'true()'"/>
<xsl:function name="local:filter" as="xs:boolean">
  <xsl:param name="e" as="element(employee)"/>
  <xsl:sequence _select="$e/({$filter})"/>
</xsl:function>
<xsl:template match="/">
  <report>
    <xsl:apply-templates mode="report" select="//employee[local:filter(.)]"/>
  </report>
</xsl:template>

```

If the supplied value of the filter parameter is, say `location = "UK"`, then the report will cover employees based in the UK.

Note:

The stylesheet function `local:filter` is used here in preference to direct use of the supplied predicate within the `select` attribute of the `xsl:apply-templates` instruction because it reduces exposure to code injection attacks. It does not necessarily eliminate all such risks, however. For example, it would be possible for a caller to supply an expression that never terminates, thus creating a denial-of-service risk.

3.14 Built-in Types

Every XSLT 3.0 processor includes the following named type definitions in the [in-scope schema components](#):

- All built-in types defined in [\[XML Schema Part 2\]](#), including `xs:anyType` and `xs:anySimpleType`.
- The following types defined in [\[XPath 3.0\]](#): `xs:yearMonthDuration`, `xs:dayTimeDuration`, `xs:anyAtomicType`, `xs:untyped`, and `xs:untypedAtomic`.

XSLT 3.0 processors MAY optionally include types defined in XSD 1.1 (see [\[XML Schema 1.1 Part 1\]](#)). XSD 1.1 adopts the types `xs:yearMonthDuration`, `xs:dayTimeDuration`, and `xs:anyAtomicType` previously defined in XPath 2.0, and adds one new type: `xs:dateTimeStamp`. XSD 1.1 also allows implementers to define additional primitive types, and XSLT 3.0 permits such types to be supported by an XSLT processor.

A [schema-aware XSLT processor](#) additionally supports:

- User-defined types, and element and attribute declarations, that are imported using an `xsl:import-schema` declaration as described in [3.15 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 Schema Documents or the Schema for Schema Documents (Datatypes). The Schema for Schema Documents, as well as defining built-in types such as `xs:integer` and `xs:double`, also defines types that are intended for use only within that schema, such as `xs:derivationControl`. A [stylesheet](#) that is designed to process XML Schema documents as its input or output may import the Schema for Schema Documents.

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 [24.1 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.

[3.15 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 [27 Conformance](#).

```
<!-- Category: declaration -->
<xsl:import-schema
  namespace? = uri
  schema-location? = uri >
<!-- 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 [3.14 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 [package](#), 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 within the [declaring package](#); importing components in one stylesheet module makes them available throughout the [package](#).

The schema components imported into different [packages](#) within a [stylesheet](#) must be consistent. Specifically, it is not permitted to use the same name in the same XSD symbol space to refer to different schema components within

different packages; and the union of the schema components imported into the packages of a stylesheet must constitute a valid schema (as well as the set of schema components imported into each package forming a valid schema in its own right).

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 one of the following must be true:

- the namespace attribute of the [xsl:import-schema](#) element and the targetNamespace attribute of the [xs:schema](#) element are both absent (indicating a no-namespace schema), or
- the namespace attribute of the [xsl:import-schema](#) element and the targetNamespace attribute of the [xs:schema](#) element are both present and both have the same value, or
- the namespace attribute of the [xsl:import-schema](#) element is absent and the targetNamespace attribute of the [xs:schema](#) element is present, in which case the target namespace is as given on the [xs:schema](#) element.

[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://example.com/ns/yes-no"`

```

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

<xsl:variable name="condition" select="local:yes-no('yes')"
  as="local:yes-no"/>

```

There are two built-in functions ([analyze-string](#)^{F030} and [json-to-xml](#)) whose result is an XML structure for which a schema is defined. The namespace for these schema definitions is (in both cases) <http://www.w3.org/2005/xpath-functions>. Schema components for these namespaces are available for reference within the stylesheet if and only if an [`xsl:import-schema`](#) declaration is present referencing this namespace. If such a declaration is present, then the schema that is imported is the schema defined in the specification of these functions: the `schemaLocation` attribute has no effect.

4 Data Model

The data model used by XSLT is the XPath 3.0 and XQuery 3.0 data model (XDM), as defined in [\[XDM 3.0\]](#). 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 [4.3 Stripping Whitespace from the Stylesheet](#) and [4.4.2 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, #xA, #xD, 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 [\[XDM 3.0\]](#) is capable of representing either an XML 1.0 document (conforming to [\[XML 1.0\]](#) and [\[Namespaces in XML\]](#)) 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 3.0 can be used with either of these XML versions.

Construction of the XDM tree is outside the scope of this specification, so XSLT 3.0 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 [26 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 3.0 and XPath 3.0, 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 (and, indeed, between different editions of XML 1.0). Implementations MAY support any version; it is RECOMMENDED that an XSLT 3.0 processor that implements the 1.1 versions SHOULD also provide a mode that supports the 1.0 versions. It is thus [implementation-defined](#) which versions and editions of XML and XML Namespaces are supported by the implementation.

Note:

The specification referenced as [\[Namespaces in XML\]](#) was actually published without a version number.

The current version of [\[XML Schema 1.1 Part 2\]](#) references the XML 1.1 specifications, but the previous version ([\[XML Schema Part 2\]](#)) (that is, XSD 1.0) remains in widespread use, and only references XML 1.0. With processors lacking support for XSD 1.1, therefore, datatypes such as `xs:NCName` and `xs:ID` may be constrained by the XML 1.0 rules, and not allow the full range of values permitted by XML 1.1. It is RECOMMENDED that implementers wishing to support XML 1.1 should consult [\[XML Schema 1.0 and XML 1.1\]](#) for guidance.

4.2 XDM versions

XSLT 3.0 REQUIRES a processor to support XDM 3.0 as defined in [\[XDM 3.0\]](#), augmented with support for maps as described in [21 Maps](#).

A processor MAY also provide a user option to support XDM 3.1 as defined in [\[XDM 3.1\]](#), in which case it must do so as defined in [27.7 XPath 3.1 Feature](#).

Note:

The essential differences between XDM 3.0 (with the extensions defined in this specification) and XDM 3.1 are that XDM 3.1 adds support for arrays, and for the `xs:numeric` union type.

A processor **MAY** also provide a user option to support versions of XDM later than 3.1, in which case the way it does so is [implementation-defined](#).

[4.3 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:
 - o The parent of the text node is not an [xsl:text](#) element
 - o 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:accumulator
xsl:analyze-string
xsl:apply-imports
xsl:apply-templates
xsl:attribute-set
xsl:call-template
xsl:character-map
xsl:choose
xsl:evaluate
xsl:fork
xsl:merge
xsl:merge-source
xsl:mode
xsl:next-iteration
xsl:next-match
xsl:override
xsl:package
xsl:stylesheet
xsl:transform
xsl:use-package

```

5. Any [whitespace text node](#) whose immediate following-sibling node is an [xsl:param](#) or [xsl:sort](#) or [xsl:context-item](#) or [xsl:on-completion](#) element is removed from the tree, regardless of any

`xml:space` attributes.

6. Any [whitespace text node](#) whose immediate preceding-sibling node is an [`xsl:catch`](#) element is removed from the tree, regardless of any `xml:space` attributes.

[ERR XTE0260] 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 whitespace text nodes in that part of the stylesheet 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.

4.4 Preprocessing Source Documents

Source documents supplied as input to a transformation may be subject to preprocessing. Two kinds of preprocessing are defined: stripping of type annotations (see [4.4.1 Stripping Type Annotations from a Source Tree](#)), and stripping of whitespace text nodes (see [4.4.2 Stripping Whitespace from a Source Tree](#)).

Stripping of type annotations happens before stripping of whitespace text nodes.

The source documents to which this applies are as follows:

- The document containing the [global context item](#) if it is a node;
- Any documents containing a node present in the [initial match selection](#);
- Any document containing a node that is returned by the functions [`document`](#), [`doc`](#)^{FO30}, or [`collection`](#)^{FO30};
- Any document read using [`xsl:source-document`](#).

Note:

This list excludes documents passed as the values of [stylesheet parameters](#) or parameters of the [initial named template](#) or [initial function](#), trees created by functions such as [`parse-xml`](#)^{FO30}, [`parse-xml-fragment`](#), [`analyze-string`](#)^{FO30}, or [`json-to-xml`](#), nor values returned from [extension functions](#).

If a node other than a document node is supplied (for example as the global context item), then the preprocessing is applied to the entire document containing that node. If several nodes within the same document are supplied (for example as nodes in the initial match selection, or as nodes returned by the [`collection`](#)^{FO30} function), then the preprocessing is only applied to that document once. If a whitespace text node is supplied (for example as the

global context item) and the rules cause this node to be stripped from its containing tree, then the behavior is as if this node had not been supplied (which may cause an error, for example if a global context item is required.)

The rules determining whether or not stripping of annotations and/or whitespace happens are defined at the level of a [package](#). Declarations within a [library package](#) only affect the handling of documents loaded using a call on the [document](#), [doc^{FO30}](#), or [collection^{FO30}](#) functions or an evaluation of an [xsl:source-document](#) instruction appearing lexically within the same package. Declarations within the [top-level package](#) also affect the processing of the [global context item](#) and the [initial match selection](#).

The semantics of the [document](#), [doc^{FO30}](#), and [collection^{FO30}](#) functions are formally defined in terms of mappings from URIs to document nodes maintained within the dynamic context (see [5.3.3 Initializing the Dynamic Context](#)). The effect of the declarations that control stripping of type annotations and whitespace is therefore to modify this mapping (so it now maps the URI to a stripped document). The modification applies to the dynamic context for calls to these function appearing within a particular package; each package therefore has a different set of mappings. This means that when two calls to the [doc^{FO30}](#) function appear in different packages, specifying the same absolute URI, then in general different documents are returned. An implementation MAY return the same document for two such calls if it is able to determine that the effect of the annotation and whitespace stripping rules in both packages is the same.

The effect of dynamic calls to the [document](#), [doc^{FO30}](#), and [collection^{FO30}](#) functions is defined in the same way as for other functions with dependencies on the dynamic context. As described in [5.3.4 Additional Dynamic Context Components used by XSLT](#), named function references (such as `doc#1`) and calls on [function-lookup^{FO30}](#) (for example, `function-lookup("doc", 1)`) are defined to retain the XPath static and dynamic context at the point of invocation as part of the closure of the resulting function item, and to use this preserved context when a dynamic function call is subsequently made using the function item.

[4.4.1 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^{DM30}](#).]

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:package](#) element. This attribute has three permitted values: `strip`, `preserve`, and `unspecified`. The default value is `unspecified`.

The `input-type-annotations` attribute may also be specified on the [xsl:stylesheet](#) element; if it is specified at this level then it must be consistent for all stylesheet modules within the same package.

[ERR XTSE0265] It is a [static error](#) if there is a [stylesheet module](#) in a [package](#) that specifies `input-type-annotations="strip"` and another [stylesheet module](#) that specifies `input-type-annotations="preserve"`, or if a stylesheet module specifies the value `strip` or `preserve` and the same value is not specified on the [xsl:package](#) element of the containing package.

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`](#)^{FO30} function.

4.4.2 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.

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.

The stripping process that applies for a particular [package](#) is determined by the [`xsl:strip-space`](#) and [`xsl:preserve-space`](#) declarations within that package.

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](#)^{XP30}; an element name matches an [xsl:strip-space](#) or [xsl:preserve-space](#) element if it matches one of the [NameTests](#)^{XP30}. An element matches a [NameTest](#)^{XP30} if and only if the [NameTest](#)^{XP30} would be true for the element as an XPath node test.

[ERR XTSE0270] It is a [static error](#) if within any [package](#) the same [NameTest](#)^{XP30} appears in both an [xsl:strip-space](#) and an [xsl:preserve-space](#) declaration if both have the same [import precedence](#). Two NameTests are considered the same if they match the same set of names (which can be determined by comparing them after expanding namespace prefixes to URIs).

Otherwise, when more than one [xsl:strip-space](#) and [xsl:preserve-space](#) element within the relevant [package](#) matches, the best matching element is determined by the best matching [NameTest](#)^{XP30}. The rules are similar to those for [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.
- If several matches have the same [default priority](#) (which can only happen if one of the NameTests takes the form `*:local` and the other takes the form `prefix:*`), then the declaration that appears last in [declaration order](#) is used.

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 [\[XDM 3.0\]](#), 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 [\[XDM 3.0\]](#), 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-idref` properties. It is these properties, rather than any [type annotation](#), that are examined by the functions [`id`](#)^{FO30} and [`idref`](#)^{FO30} described in [\[Functions and Operators 3.0\]](#).

[4.6 Data Model for Streaming](#)

[4.6.1 Streamed Documents](#)

The data model for nodes in a document that is being streamed is no different from the standard XDM data model, in that it contains the same objects (nodes) with the same properties and relationships. The facilities for streaming do not change the data model; instead they impose rules that limit the ability of stylesheets to navigate the data model.

A useful way to visualize streaming is to suppose that at any point in time, there is a current position in the streamed input document which may be the start or end of the document, the start or end tag of an element, or a text, comment, or processing instruction node. From this position, the stylesheet has access to the following information:

- Properties intrinsic to the node, such as its name, its base URI, its type annotation, and its `is-id` and `is-idref` properties.
- The ancestors of the node (but navigation downwards from the ancestors is not permitted).
- The attributes of the node, and the attributes of its ancestors. For each such attribute, all the properties of the node including its string value and typed value are available, but there are limitations that restrict navigation from the attribute node to other nodes in the document.
- The in-scope namespace bindings of the node.
- In the case of attributes, text nodes, comments, and processing instructions, the string value and typed value of the node.
- In the case of element nodes, whether or not the element has children. This information is obtained by calling the [`has-children`](#)^{FO30} function. This implies that the processor performs look-ahead (limited to a single token) to determine whether the start tag is immediately followed by a matching end tag.
- In the case of document nodes, details of unparsed entities in the document. This information is obtained by calling the [`unparsed-entity-uri`](#) and [`unparsed-entity-public-id`](#) functions. A processor might enable this by reading the DTD as soon as the document is opened. Since comments and processing instructions that precede the DOCTYPE declaration are available as children of the document node, this also implies that a streaming processor needs sufficient memory to hold these comments and processing instructions until the start tag of the first element is encountered. Information about unparsed entities remains available for the duration of processing, in the same way as attributes of ancestor elements.

The children and other descendants of a node are not accessible except as a by-product of changing the current position in the document. The same applies to properties of an element or document node that require examination of the node's descendants, that is, the string value and typed value. This is enforced by means of a rule that only one expression requiring downward navigation from a node is permitted.

Information about the type of a node is in general considered a property intrinsic to the node, and is available without advancing the input stream. There is an exception for an expression of the form `(/) instance of document-node(element(invoice))`. This is not guaranteed streamable, because it requires reading ahead to check that the document node has only one element child. However, a processor that knows that the parser delivering the document stream is only capable of delivering well-formed documents may use this knowledge (along with the limited look-ahead needed to get the name of the outermost element) to make this expression streamable.

A streaming processor is not required to read any more of the source document than is needed to generate correct stylesheet output. It is not required to read the full source document merely in order to satisfy the requirement imposed by the XML Recommendation that an XML Processor must report violations of well-formedness in the input.

More detailed rules are defined in [19 Streamability](#).

4.6.2 Other Data Structures

Two new data structures have been added to the data model: maps and arrays. Both are defined in XPath 3.1, but maps are also available in XSLT processors that only support XPath 3.0 (see [21 Maps](#)).

Streaming facilities in this specification are, for the most part, relevant only to streamed processing of XML trees, and not to other structures such as sequences, maps and arrays, which will typically be held in memory unless the processor is capable of avoiding this.

Maps, however, play an important role in enabling streamed applications to be written. For example, a map can be used as the data structure maintained by an accumulator (see [18.2 Accumulators](#)) to remember information that has been retrieved from a streamed document, given that it is not possible to revisit the same nodes later. There is also a special streamability rule for map constructor expressions (see [21.6 Maps and Streaming](#)) that allows such an expression to make multiple downward selections in the streamed input document: for example one can write `map{'authors':data(author), 'editors':data(editor)}`, which gathers the values of these two elements, or sets of elements, from the input stream, regardless what order they appear in — even if they are interleaved.

The rules for creating maps and arrays are designed to ensure that the entries in a map, and the members of an array, cannot contain nodes from a streamed document. This is achieved by the way in which the streamability properties of the relevant expressions and functions are defined.

By contrast, sequences can and often do contain nodes from streamed documents, and a major purpose of the rules for streamability is to make this possible.

4.7 Limits

The XDM data model (see [\[XDM 3.0\]](#)) leaves it to the host language to define limits. This section describes the limits that apply to XSLT.

Limits on some primitive datatypes 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.

[4.8 Disable Output Escaping](#)

For backwards compatibility reasons, XSLT 3.0 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 [26.1 Character Maps](#)) was 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 [26.2 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.

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 Names

5.1.1 Qualified Names

Many constructs appearing in a stylesheet, for example [named templates](#), [modes](#), and [attribute sets](#), are named using a qualified name: this consists of a local name and an optional namespace URI.

In most cases where such names are written in a [stylesheet](#), the syntax for expressing the name is given by the production [EQName^{XP30}](#) in the XPath specification. In practice, this means that three forms are permitted:

- A simple NCName appearing on its own (without any prefix). This represents the local name of the object. The interpretation of unprefixed names is described below.
- A [lexical QName](#) written in the form NCName ":" NCName where the first part is a namespace prefix and the second part is the local name. The namespace part of the object's name is then derived from the prefix by examining the in-scope namespace bindings of the element node in the stylesheet where the name appears.
- A [URIQualifiedNames](#)^{XP30} in the form "Q{" URI? "}" NCName where the two parts of the name, that is the namespace part and the local part, both appear explicitly. If the URI part is omitted (for example Q{}local), the resulting expanded QName is a QName whose namespace part is absent.

Note:

There are a few places where the third form, a URIQualifiedName, is not permitted. These include the name attribute of [xsl:element](#) and [xsl:attribute](#) (which have a separate namespace attribute for the purpose), and constructs defined by other specifications. For example, names appearing within an embedded xs:schema element must follow the XSD rules.

[**DEFINITION:** An **expanded QName** is a value in the value space of the xs:QName datatype as defined in the XDM data model (see [\[XDM 3.0\]](#)): that is, a triple containing namespace prefix (optional), namespace URI (optional), and local name. 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.]

[**DEFINITION:** An **EQName** is a string representing an [expanded QName](#) where the string, after removing leading and trailing whitespace, is in the form defined by the [EQName^{XP30}](#) production in the XPath specification.]

[**DEFINITION:** A **lexical QName** is a string representing an [expanded QName](#) where the string, after removing leading and trailing whitespace, is within the lexical space of the xs:QName datatype as defined in XML Schema (see [\[XML Schema Part 2\]](#)): that is, a local name optionally preceded by a namespace prefix and a colon.]

Note that every [lexical QName](#) is an [EQName](#), but the converse is not true.

The following rules are used when interpreting a [lexical QName](#):

1. [**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 an attribute or text node within a stylesheet module, or as the result of evaluating an XPath expression contained in such a node. The element containing this attribute or text node is referred to as the **defining element** of the lexical QName.]

2. If the lexical 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](#)^{DM30}) is *not* used for unprefixed names.

[ERR XTSE0280] In the case of a prefixed [lexical QName](#) used as the value (or as part of 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 [lexical QName](#).

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

3. If the lexical QName has no prefix, then:

a. In the case of an unprefixed QName used as a [NameTest](#) within an XPath [expression](#) (see [5.2 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 [5.1.2 Unprefixed Lexical QNames in Expressions and Patterns](#).

b. If the name is in one of the following categories, then the default namespace of the [defining element](#) is used:

i. 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).

ii. The default namespace is used when expanding the first argument of the function [element-available](#).

iii. The default namespace applies to any unqualified element names appearing in the `cdata-section-elements` or `suppress-indentation` attributes of [xsl:output](#) or [xsl:result-document](#)

c. In all other cases, a [lexical QName](#) with no prefix represents an [expanded QName](#) in no namespace (that is, an `xs:QName` value in which both the prefix and the namespace URI are absent).

[5.1.2 Unprefixed Lexical QNames in Expressions and Patterns](#)

The attribute `[xsl:]xpath-default-namespace` (see [3.4 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 or text node of that element (including XPath expressions in [attribute value templates](#) and [text value templates](#)). The effect of this is specified in [\[XPath 3.0\]](#); 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](#)^{DM30}) 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.

5.1.3 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 [\[Functions and Operators 3.0\]](#) and for standard functions defined in this specification.]
- The namespace `http://www.w3.org/2005/xpath-functions/math` is used for mathematical functions in the function library defined in [\[Functions and Operators 3.0\]](#).
- The namespace `http://www.w3.org/2005/xpath-functions/map` is used for functions defined in this specification relating to the manipulation of [maps](#).
- The namespace `http://www.w3.org/2005/xpath-functions/array` is reserved for use as described in [\[Functions and Operators 3.1\]](#). The namespace is reserved whether or not the processor actually supports XPath 3.1.
- [**DEFINITION:** The **XML namespace**, defined in [\[Namespaces in XML\]](#) 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.]
- [DEFINITION: The **standard error namespace** `http://www.w3.org/2005/xqt-errors` is used for error codes defined in this specification and related specifications. It is also used for the names of certain predefined variables accessible within the scope of an [xsl:catch](#) element.]
- The namespace `http://www.w3.org/2000/xmlns/` is reserved for use as described in [\[Namespaces in XML\]](#). No element or attribute node can have a name in this namespace, and although the prefix `xmlns` is implicitly bound to this namespace, no namespace node will ever define this binding.

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.

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](#), nor in the names of [extension functions](#) or [extension instructions](#).

It is not an error to use a reserved namespace in the name of an [extension attribute](#): attributes such as `xml:space` and `xsi:type` fall into this category. XSLT processors MUST NOT reject such attributes, and MUST NOT attach any meaning to them other than any meaning defined by the relevant specification.

[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](#), an [accumulator](#), or a [character map](#); except that the name `xsl:initial-template` is permitted as a template name.

Note:

The name `xsl:original` is used within [xsl:override](#) to refer to a [component](#) that is being overridden. Although the name `xsl:original` is used to refer to the component, the component has its own name, and no component ever has the name `xsl:original`.

5.2 Expressions

XSLT uses the expression language defined by XPath 3.0 [\[XPath 3.0\]](#). 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^{XP30}](#) defined in [[XPath 3.0](#)], with the extensions defined in [21 Maps](#).]

If the processor implements the [XPath 3.1 Feature](#), then the definition of the production Expr from XPath 3.1 is used.

If the processor is configured to use a version of XPath later than XPath 3.1, then the syntax of an XPath expression is [implementation-defined](#).

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

Except where [forwards compatible behavior](#) is enabled (see [3.10 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](#) or [text value template](#), does not match the XPath production [Expr^{XP30}](#), 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 3.0](#)].

The transformation fails with a [dynamic error](#) if any XPath [expression](#) is evaluated and raises a dynamic error. Error codes are defined in [[XPath 3.0](#)].

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 3.0](#)].

[**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:** When used in this specification without further qualification, the term **function conversion rules** means the function conversion rules defined in [[XPath 3.0](#)], applied with XPath 1.0 compatibility mode set to false.]

Note:

These are the rules defined in [[XPath 3.0](#)] for converting the supplied argument of a function call to the required type of that argument, as defined in the function signature. The same rules are used in XSLT for converting the value of a variable to the declared type of the variable, or the result of evaluating a function or template body to the declared type of the function or template. They are also used when parameters are supplied to a template using [xsl:with-param](#). In all such cases, the rules that apply are the XPath 3.0 rules without XPath 1.0 compatibility mode. The rules with XPath 1.0 compatibility mode set to true are used only for XPath function calls, and for the operands of certain XPath operators.

This specification also invokes the XPath 3.0 [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.

The XPath specification states that the host language must specify whether the XPath processor normalizes all line breaks on input, before parsing, and if it does so, whether it uses the rules of [XML 1.0] or [XML 1.1]. In the case of XSLT, all handling of line breaks is the responsibility of the XML parser (which may support either XML 1.0 or XML 1.1); the XSLT and XPath processors perform no further changes.

Note:

Most XPath expressions in a stylesheet appear within XML attributes. They are therefore subject to XML line-ending normalization (for example, a CRLF sequence is normalized to LF) and also to XML attribute-value normalization, which replaces tabs and newlines by spaces. XPath expressions appearing in text value templates, however (see [5.6.2 Text Value Templates](#)) are subject to line-ending normalization but not attribute-value normalization. In both cases, normalization of whitespace can be prevented by using character references such as `
`.

5.3 The Static and Dynamic Context

XPath defines the concept of an [expression context^{XP30}](#) which contains all the information that can affect the result of evaluating an [expression](#). The expression context has two parts, the [static context^{XP30}](#), and the [dynamic context^{XP30}](#). The components that make up the expression context are defined in the XPath specification (see [Section 2.1 Expression Context^{XP30}](#)). 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:

[5.3.1 Initializing the Static Context](#)

[5.3.2 Additional Static Context Components used by XSLT](#)

[5.3.3 Initializing the Dynamic Context](#)

[5.3.4 Additional Dynamic Context Components used by XSLT](#)

5.3.1 Initializing the Static Context

The [static context](#)^{XP30} 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 or text node 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 is processed with [XSLT 1.0 behavior](#) (see [3.9 Backwards Compatible Processing](#)).
- The [statically known namespaces](#)^{XP30} are the namespace declarations that are in scope for the containing element.
- The [default element/type namespace](#)^{XP30} is the namespace defined by the `[xsl:]xpath-default-namespace` attribute on the innermost enclosing element that has such an attribute, as described in [5.1.2 Unprefixed Lexical 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](#)^{XP30} is the [standard function namespace](#), defined in [\[Functions and Operators 3.0\]](#). 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 functions in this namespace.
- The [in-scope schema definitions](#)^{XP30} for the XPath expression are the same as the [in-scope schema components](#) for the [stylesheet](#), and are as specified in [3.14 Built-in Types](#).
- The [in-scope variables](#)^{XP30} are defined by the [variable binding elements](#) that are in scope for the containing element (see [9 Variables and Parameters](#)).
- The [context item static type](#)^{XP30} may be determined by an XSLT processor that performs static type inferencing, using rules that are outside the scope of this specification; if no static type inferencing is done, then the context item static type for every XPath expression is `item()`. Note that some limited static type inferencing is required in the case of a processor that performs streamability analysis: see [19.1 Determining the Static Type of a Construct](#).
- The [statically known function signatures](#)^{XP30} are:
 - The functions defined in [\[Functions and Operators 3.0\]](#) in namespaces `http://www.w3.org/2005/xpath-functions` and `http://www.w3.org/2005/xpath-functions/math`;
 - The functions defined in this specification in namespaces `http://www.w3.org/2005/xpath-functions` and `http://www.w3.org/2005/xpath-functions/map`;
 - Constructor functions for all the simple types in the [in-scope schema definitions](#)^{XP30}, including both built-in types and user-defined types;
 - The [stylesheet functions](#) defined in the containing [package](#);
 - Stylesheet functions defined in used packages, subject to visibility: see [3.5.2 Dependencies between Packages](#);
 - any [extension functions](#) bound using [implementation-defined](#) mechanisms (see [24 Extensibility and Fallback](#)).

Note:

The term [extension function](#) includes both vendor-supplied and user-written extension functions.

Note:

It follows from the above that a conformant XSLT processor must implement the entire library of functions defined in [\[Functions and Operators 3.0\]](#) as well as those defined in this specification.

- The [statically known collations](#)^{XP30} are [implementation-defined](#), except that they MUST always include (a) the Unicode codepoint collation, defined in [Section 5.3 Comparison of strings](#)^{FO30}, and (b) the family of UCA collations described in [13.4 The Unicode Collation Algorithm](#).
- The [default collation](#)^{XP30} is defined by the value of the `[xsl:]default-collation` attribute on the innermost enclosing element that has such an attribute. For details, see [3.7.1 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](#).
- **Static base URI:** In a conventional interpreted environment, the static base URI of an expression in the stylesheet is the base URI of the containing element in the stylesheet. The concept of the base URI of a node is defined in [Section 5.2 base-uri Accessor](#)^{DM30}.

When stylesheets are executed in an environment where no source code is present (for example, because the code of the stylesheet has been compiled and is distributed as executable object code), it is RECOMMENDED (subject to operational constraints such as security) that the static base URI used during stylesheet evaluation should be the location from which the stylesheet was loaded for execution (its “deployed location”). This means, for example, that when the [doc](#)^{FO30} or [document](#) functions are called with a relative URI, the required document is by default located relative to the deployed location of the stylesheet.

Whether or not the stylesheet is executed directly from source code, it is possible that no static base URI is available, for example because the code was supplied as an anonymous input stream, or because security policies are set to prevent executable code discovering the location from which it was loaded. If the static base URI is not known, the [static-base-uri](#)^{FO30} function returns an empty sequence, and other operations that depend on the static base URI may fail with a dynamic error.

- The set of [statically known documents](#)^{XP30} is [implementation-defined](#).
- The set of [statically known collections](#)^{XP30} is [implementation-defined](#).
- The [statically known default collection type](#)^{XP30} is [implementation-defined](#).
- The set of [statically known decimal formats](#)^{XP30} is the set of decimal formats defined by [xsl:decimal-format](#) declarations in the stylesheet.

Note:

XSLT 3.0 provides support for the `exponent-separator` property which is added to the static context in XPath 3.1; when XSLT 3.0 is used with XPath 3.0, this property is ignored.

5.3.2 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 values of system properties, used by the [`system-property`](#) function
- The set of available instructions, used by the [`element-available`](#) function

A dynamic function call clears the first of these components: this means that a dynamic call to the [`key`](#) function will always raise a dynamic error (the key name is unknown). The values of system properties and the set of available instructions, by contrast, reflect the capabilities and configuration of the processor rather than values specific to the stylesheet code itself; the result of a dynamic call to [`system-property`](#) or [`element-available`](#) will reflect the information available to the processor at evaluation time.

Note:

If these functions are called within a [`static expression`](#), the results will reflect the capabilities and configuration of the processor used to perform static analysis, while if they are called elsewhere, the results should reflect the capabilities and configuration of the processor used to perform dynamic evaluation, which might give a different result. These calls should not be pre-evaluated at compile time unless it is known that this will give the same result.

5.3.3 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 [\[Functions and Operators 3.0\]](#) are defined to be [`deterministic`](#)^{FO30}, meaning that if they are called twice during the same [`execution scope`](#)^{FO30}, with the same arguments, then they return the same results (see [Section 1.6 Terminology](#)^{FO30}). In XSLT, the execution of a stylesheet defines the execution scope. This means, for example, that if the function [`current-dateTime`](#)^{FO30} 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](#)^{XP30} must be stable: *function implementations, currentDateTime, 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 [5.3.4 Additional Dynamic Context Components used by XSLT](#) are also *not* stable.

As specified in [\[Functions and Operators 3.0\]](#), implementations may provide user options that relax the requirement for the [`doc`](#)^{FO30} and [`collection`](#)^{FO30} 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 [3.13.1 Conditional Element Inclusion](#).

[**DEFINITION:** A component of the context that has no value is said to be **absent**.] This is a distinguishable state, and is not the same as having the empty sequence as its value.

5.3.3.1 [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 [\[XDM 3.0\]](#)) is either an atomic value (such as an integer, date, or string), a node, or a function item. 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 or a function item, 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.

The context item for evaluating global variables in the [top-level package](#) is set to the [global context item](#) supplied when the transformation is invoked (see [2.3 Initiating a Transformation](#)). In [library packages](#), the context item for evaluating global variables is [absent](#).

For an XPath expression contained in a [value template](#), 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 [sequence constructor](#).

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 [20.4.1 fn: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 [absent](#).

When the focus is [absent](#), evaluation of any [expression](#) that references the context item, context position, or context size results in a [dynamic error \[ERR_XPDY0002\]](#)^{XP30}

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 an item J has the [context item](#) (and therefore the [context node](#), if J is a node) set to J , and the [context position](#) and [context size](#) both set to 1 (one).]

5.3.3.2 [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](#)^{XP30} of an XPath expression are initialized.

- The [dynamic variables](#)^{XP30} are the current values of the in-scope [variable binding elements](#).
- The [named functions](#)^{XP30} (representing the functions accessible using [function-available](#) or [function-lookup](#)^{F030}) include all the functions available in the static context, and may also include an additional [implementation-defined](#) set of functions that are available dynamically but not statically.

Note:

This set therefore includes some functions that are not available for dynamic calling using [xsl:evaluate](#), for example [stylesheet functions](#) whose visibility is private, and XSLT-defined functions such as [current](#) and [key](#).

Note:

The rule that all functions present in the static context must always be present in the dynamic context is a consistency constraint. The effect of violating a consistency constraint is [implementation-defined](#): it does not necessarily lead to an error. For example, if the version of a used package that is available at evaluation time does not include all public user-defined functions that were available in the version that was used at analysis time, then a processor MAY recover by signaling an error only if the function is actually called. Conversely, if the evaluation-time version of the package includes additional public functions, these MAY be included in the dynamic context even though they were absent from the static context. Dynamic calling of functions using [function-lookup^{FO30}](#) may therefore be an effective strategy for coping with variations between versions of a library package on which a stylesheet depends.

- The [available documents^{XP30}](#) are defined as part of the XPath 3.0 dynamic context to support the [doc^{FO30}](#) function, but this component is also referenced by the similar XSLT [document](#) function: see [20.1 fn:document](#). This variable defines a mapping between URIs passed to the [doc^{FO30}](#) or [document](#) function and the document nodes that are returned.

The mapping from URIs to document nodes is affected by [xsl:strip-space](#) declarations and by the [input-type-annotations](#) attribute, and may therefore vary from one package to another.

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^{XP30}](#) 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.

- All other aspects of the dynamic context (for example, the current date and time, the implicit timezone, the default language, calendar, and place, the available documents, text resources, and collections, and the default collection — details vary slightly between XPath 3.0 and XPath 3.1) are [implementation-defined](#), and do not change in the course of a single transformation, except to the extent that they MAY be different from one [package](#) to another.

[5.3.4 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 [6.8 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 [6.6 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 [14.2.1 fn:current-group](#) and [14.2.2 fn:current-grouping-key](#);

Note:

In XSLT 3.0 the initial value of these two properties is “absent”, which means that any reference to their values causes a dynamic error. Previously, the initial value was an empty sequence.

- The [current merge group](#) and [current merge key](#), which provide information about the collection of items currently being processed by an [xsl:merge](#) instruction.
- 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 [17.2 fn:regex-group](#).
- The [output state](#): this is a flag whose two possible values are [final output state](#) and [temporary output state](#). The initial setting when the stylesheet is invoked by executing a template is [final output state](#), and it is switched to [temporary output state](#) by instructions such as [xsl:variable](#). For more details, see [25.2 Restrictions on the use of xsl:result-document](#).
- The [current output URI](#): this is the URI associated with the result tree to which instructions are currently writing. The current output URI is initially the same as the [base output URI](#). During the evaluation of an [xsl:result-document](#) instruction, the current output URI is set to the absolute URI identified by the [href](#) attribute of that instruction.

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.

Components of the Dynamic Evaluation Context

Component	Initial Setting	Set by	Cleared by
focus	See 2.3 Initiating a Transformation .	xsl:apply-templates , xsl:for-each , xsl:for-each-group , xsl:analyze-string , evaluation of patterns	Calls to stylesheet functions

Component	Initial Setting	Set by	Cleared by
<u>current</u> <u>template</u> <u>rule</u>	If apply-templates invocation is used (see 2.3.3 Apply-Templates Invocation), then for each item in the <u>initial match selection</u> , the <u>current template rule</u> is initially set to the template rule chosen for processing that item. Otherwise, <u>absent</u> .	<u>xsl:apply-templates</u> , <u>xsl:apply-imports</u> , <u>xsl:next-match</u>	See 6.8 Overriding Template Rules .
<u>current mode</u>	the initial <u>mode</u>	<u>xsl:apply-templates</u>	Calls to <u>stylesheet functions</u> . Also cleared while evaluating global variables and stylesheet parameters, <u>patterns</u> , and the sequence constructor contained in <u>xsl:key</u> or <u>xsl:sort</u> . Clearing the current mode causes the current mode to be set to the default (unnamed) mode.
<u>current group</u>	absent	<u>xsl:for-each-group</u>	See 14.2.1 fn:current-group .
<u>current grouping key</u>	absent	<u>xsl:for-each-group</u>	See 14.2.2 fn:current-grouping-key .
<u>current merge group</u>	absent	<u>xsl:merge</u>	See 15.6.1 fn:current-merge-group .
<u>current merge key</u>	absent	<u>xsl:merge</u>	See 15.6.2 fn:current-merge-key .
<u>current captured substrings</u>	empty sequence	<u>xsl:matching-substring</u>	<u>xsl:non-matching-substring</u> ; Calls to <u>stylesheet functions</u> , dynamic function calls, evaluation of global variables, stylesheet parameters, and <u>patterns</u>

Component	Initial Setting	Set by	Cleared by
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
current output URI	base output URI	xsl:result-document	Calls to stylesheet functions , dynamic function calls, evaluation of global variables , stylesheet parameters , and patterns .

[DEFINITION: The **initial setting** of a component of the dynamic context is used when evaluating [global variables](#) and [stylesheet parameters](#), when evaluating the use and match attributes of [xsl:key](#), and when evaluating the initial-value of [xsl:accumulator](#) and the select expressions or contained sequence constructors of [xsl:accumulator-rule](#)].

[DEFINITION: The term **non-contextual function call** is used to refer to function calls that do not pass the dynamic context to the called function. This includes all calls on [stylesheet functions](#) and all [dynamic function invocations](#)^{XP30}, (that is calls to function items as permitted by XPath 3.0). It excludes calls to some functions in the namespace <http://www.w3.org/2005/xpath-functions>, in particular those that explicitly depend on the context, such as the [current-group](#) and [regex-group](#) functions. It is [implementation-defined](#) whether, and under what circumstances, calls to [extension functions](#) are non-contextual.]

Named function references (such as `position#0`) and calls on [function-lookup](#)^{FO30} (for example, `function-lookup("position", 0)`) are defined to retain the XPath static and dynamic context at the point of invocation as part of the closure of the resulting function item, and to use this preserved context when a dynamic function call is subsequently made using the function item. This rule does not extend to the XSLT extensions to the dynamic context defined in this section. If a dynamic function call is made that depends on the XSLT part of the dynamic context (for example, `regex-group#1(2)`), then the relevant components of the context are cleared as described in the table above.

5.4 Defining a Decimal Format

The definition of the [format-number](#)^{FO30} function is now in [\[Functions and Operators 3.0\]](#). What remains here is the definition of the [xsl:decimal-format](#) declaration, which provides the context for this function when used in an XSLT stylesheet.

```
<!-- Category: declaration -->
<xsl:decimal-format
  name? = eqname
  decimal-separator? = char
  grouping-separator? = char
  infinity? = string
  minus-sign? = char
  exponent-separator? = char
  NaN? = string
  percent? = char
  per-mille? = char
  zero-digit? = char
  digit? = char
  pattern-separator? = char />
```

The [xsl:decimal-format](#) element sets the **statically known decimal formats** component of the static context for XPath expressions, which controls the interpretation of a [picture string](#) used by the [format-number^{FO30}](#) function.

[**DEFINITION:** The **picture string** is the string supplied as the second argument of the [format-number^{FO30}](#) function.]

Note:

The [format-number^{FO30}](#) function, previously defined in this specification, is now defined in [\[Functions and Operators 3.0\]](#).

A [package](#) 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 package 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.]

The attributes of the [xsl:decimal-format](#) declaration define the value of the corresponding property in the relevant decimal format in the [statically known decimal formats^{XP30}](#) component of the static context for all XPath expressions in the package. The attribute names used in the XSLT 3.0 syntax are the same as the property names used in the definition of the static context.

The `exponent-separator` attribute is provided for use with XPath 3.1. It has no effect when used with XPath 3.0.

The scope of an [xsl:decimal-format](#) name is the package in which it is declared; the name is available for use only in calls to [format-number^{FO30}](#) that appear within the same package.

If a [package](#) 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^{FO30}](#) function. An outline of the purpose of each attribute is given

below; however, the definitive explanations are given as part of the specification of [format-number^{FO30}](#).

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^{FO30}](#) 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 to separate the integer part from the fractional part of the formatted number; the default value is the period character (.)
- **grouping-separator** specifies the character typically used as a thousands separator; the default value is the comma character (,)
- **percent** specifies the character used to indicate that the number is represented as a per-hundred fraction; the default value is the percent character (%)
- **per-mille** specifies the character used to indicate that the number is represented as a per-thousand fraction; the default value is the Unicode per-mille character (#x2030)
- **zero-digit** specifies the character used to represent the digit zero; the default value is the Western 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^{FO30}](#) function. In each case the value **MUST** be a single character [see [ERR XTSE0020](#)].

- **digit** specifies the character used in the [picture string](#) as a place-holder for an optional digit; the default value is the number sign character (#)
- **pattern-separator** specifies the character used to separate 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 to represent the `xs:double` value INF; the default value is the string Infinity

- `NaN` specifies the string used to represent the `xs:double` value `NaN` (not-a-number); the default value is the string `NaN`
- `minus-sign` specifies the character used to signal a negative number; the default value is the hyphen-minus character (`-`, `#xD`). 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*.

Every (named or unnamed) decimal format defined in a [package](#) is added to the [statically known decimal formats](#)^{XP30} in the [static context](#)^{XP30} of every expression in the [package](#), excluding expressions appearing in `[xsl:]use-when` attributes.

5.5 Patterns

In XSLT 3.0, patterns can match any kind of item: atomic values and function items as well as nodes.

A [template rule](#) identifies the items 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 [20.2 Keys](#)).

[**DEFINITION:** A **pattern** specifies a set of conditions on an item. An item that satisfies the conditions matches the pattern; an item that does not satisfy the conditions does not match the pattern.]

There are two kinds of pattern: [predicate patterns](#), and [selection patterns](#):

- [**DEFINITION:** A **predicate pattern** is written as `.` (dot) followed by zero or more predicates in square brackets, and it matches any item for which each of the predicates evaluates to `true`.]

The detailed semantics are given in [5.5.3 The Meaning of a Pattern](#). This construct can be used to match items of any kind (nodes, atomic values, and function items). For example, the pattern `.[starts-with(., '$')]` matches any string that starts with the character `"$"`, or a node whose atomized value starts with `"$"`. This example shows a predicate pattern with a single predicate, but the grammar allows any number of predicates (zero or more).

- [**DEFINITION:** A **selection pattern** uses a subset of the syntax for path expressions, and is defined to match a node if the corresponding path expression would select the node. Selection patterns may also be formed by combining other patterns using union, intersection, and difference operators.]

The syntax for selection patterns (`UnionExprP` in the grammar: see [5.5.2 Syntax of Patterns](#)) is a subset of the syntax for [expressions](#). Selection patterns are used only for matching nodes; an item other than a node will never match a selection pattern. As explained in detail below, a node matches a selection pattern if the node can be selected by deriving an equivalent expression, and evaluating this expression with respect to some possible context.

Note:

The specification uses the phrases *an item matches a pattern* and *a pattern matches an item* interchangeably. They are equivalent: an item matches a pattern if and only if the pattern matches the item.

5.5.1 Examples of Patterns

Example: Patterns

Here are some examples of patterns:

- `.` matches any item.
- `*` matches any element.
- `para` matches any `para` 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.
- `namespace-node()` matches any namespace 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 in a tree that is rooted at a document 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.
- `$xyz` matches any node that is present in the value of the variable `$xyz`.
- `$xyz/*` matches any element that is a descendant of a node that is present in the value of the variable `$xyz`.
- `doc('product.xml')/*` matches any element within the document whose document URI is `'product.xml'`.
- `. [. instance of node()]` matches any node. (Note the distinction from the pattern `node()`.)
- `. [. instance of xs:date]` matches any atomic value of type `xs:date` (or a type derived by restriction from `xs:date`).

- `.[. gt current-date()]` matches any date in the future. It can match an atomic value of type `xs:date` or `xs:untypedAtomic`, or a node whose atomized value is an `xs:date` or `xs:untypedAtomic` value.
- `.[starts-with(., 'e')]` matches any node or atomic value that after conversion to a string using the function conversion rules starts with the letter 'e'.
- `.[. instance of function(*)]` matches any function item.
- `.[$f(.)]` matches any item provided that the call on the function bound to the variable `$f` returns a result whose effective boolean value is true.

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 [Pattern30](#).

The grammar for patterns uses the notation defined in [Section A.1.1 Notation^{XP30}](#).

The lexical rules for patterns are the same as the lexical rules for XPath expressions, as defined in [Section A.2 Lexical structure^{XP30}](#). Comments are permitted between tokens, using the syntax `(: . . . :)`. All other provisions of the XPath grammar apply where relevant, for example the rules for whitespace handling and extra-grammatical constraints.

Patterns

[1]	<code>Pattern30</code>	<code>::= PredicatePattern UnionExprP</code>
[2]	<code>PredicatePattern</code>	<code>::= ." PredicateList^{XP30}</code>
[3]	<code>UnionExprP</code>	<code>::= IntersectExceptExprP ((union" " ") IntersectExceptExprP)*</code>
[4]	<code>IntersectExceptExprP</code>	<code>::= PathExprP ("intersect" "except") PathExprP)*</code>
[5]	<code>PathExprP</code>	<code>::= RootedPath ("/" RelativePathExprP?) ("// RelativePathExprP) RelativePathExprP</code>
[6]	<code>RootedPath</code>	<code>::= (VarRef^{XP30} FunctionCallP) PredicateList^{XP30} ("/" "//") RelativePathExprP)?</code>
[7]	<code>FunctionCallP</code>	<code>::= OuterFunctionName ArgumentListP</code>
[8]	<code>OuterFunctionName</code>	<code>::= "doc" "id" "element-with-id" "key" "root" URIQualifiedname^{XP30}</code>
[9]	<code>ArgumentListP</code>	<code>::= "(" (ArgumentP (," ArgumentP))*? ")"</code>
[10]	<code>ArgumentP</code>	<code>::= VarRef^{XP30} Literal^{XP30}</code>
[11]	<code>RelativePathExprP</code>	<code>::= StepExprP ("/" "//") StepExprP)*</code>
[12]	<code>StepExprP</code>	<code>::= PostfixExprP AxisStepP</code>
[13]	<code>PostfixExprP</code>	<code>::= ParenthesizedExprP PredicateList^{XP30}</code>

```
[14] ParenthesizedExprP ::= "(" UnionExprP ")"
[15] AxisStepP ::= ForwardStepP PredicateListXP30
[16] ForwardStepP ::= (ForwardAxisP NodeTestXP30) | AbbrevForwardStepXP30
[17] ForwardAxisP ::= ("child" ":::")
| ("descendant" ":::")
| ("attribute" ":::")
| ("self" ":::")
| ("descendant-or-self" ":::")
| ("namespace" ":::")
```

The names of these constructs are chosen to align with the XPath 3.0 grammar. Constructs whose names are suffixed with P are restricted forms of the corresponding XPath 3.0 construct without the suffix. Constructs labeled with the suffix “XP30” are defined in [\[XPath 3.0\]](#).

Where the XSLT 3.0 processor implements the [XPath 3.1 Feature](#), the definitions that apply to constructs labeled with the suffix “XP30” are those in [\[XPath 3.1\]](#).

In a [FunctionCallP](#), the EQName used for the function name must have local part `doc`, `id`, `element-with-id`, `key`, or `root`, and must use the [standard function namespace](#) either explicitly or implicitly.

In the case of a call to the [root^{FO30}](#) function, the argument list must be empty: that is, only the zero-arity form of the function is allowed.

Note:

As with XPath expressions, the pattern `/ union /*` can be parsed in two different ways, and the chosen interpretation is to treat `union` as an element name rather than as an operator. The other interpretation can be achieved by writing `(/) union /*`

5.5.3 [The Meaning of a Pattern](#)

The meaning of a pattern is defined formally as follows, where “if” is to be read as “if and only if”.

If the pattern is a [PredicatePattern PP](#), then it matches an item J if the XPath expression taking the same form as `PP` returns a non-empty sequence when evaluated with a [singleton focus](#) based on J.

Note:

The pattern `.`, which is a [PredicatePattern](#) with an empty [PredicateList^{XP30}](#), matches every item.

A predicate with the numeric value 1 (one) always matches, and a predicate with any other numeric value never matches. Numeric predicates in a [PredicatePattern](#) are therefore not useful, but are defined this way in the interests of consistency with XPath.

Otherwise (the pattern is a selection pattern), the pattern is converted to an [expression](#), called the **equivalent expression**. The equivalent expression to a [Pattern](#) is the XPath expression that takes the same lexical form as the [Pattern](#) as written, with the following adjustment:

- If any `PathExprP` in the `Pattern` is a `RelativePathExprP`, then the first `StepExprP` `PS` of this `RelativePathExprP` is adjusted to allow it to match a parentless element, attribute, or namespace node. The adjustment depends on the axis used in this step, whether it appears explicitly or implicitly (according to the rules of [Section 3.3.5 Abbreviated Syntax^{XP30}](#)), and is made as follows:
 1. 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`.
 2. 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.
 3. If `PS` uses the attribute axis (explicitly or implicitly), 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.
 4. If `PS` uses the namespace axis (explicitly or implicitly), then the axis in step `PS` is replaced by `namespace-or-top`, which is defined as follows. If the context node is a namespace node with no parent, then the `namespace-or-top` axis selects the context node; otherwise it selects the namespace nodes of the context node. It is a forwards axis whose principal node kind is namespace.

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

Note:

The purpose of this adjustment 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. To match any node at all, XSLT 3.0 allows the pattern `. [. instance of node()]` to be used.

The meaning of the pattern is then defined in terms of the semantics of the equivalent expression, denoted below as `EE`.

Specifically, an item `N` matches a pattern `P` if the following applies, where `EE` is the **equivalent expression** to `P`:

1. `N` is a node, and the result of evaluating the expression `root(.)//(EE)` with a [singleton focus](#) based on `N` is a sequence that includes the node `N`

If a pattern appears in an attribute of an element that is processed with [XSLT 1.0 behavior](#) (see [3.9 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.

Example: The Semantics of Selection Patterns

The [selection 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 [selection 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.

The [selection pattern](#) `$V` matches all nodes selected by the expression `root(.)//($V)`, that is, all nodes in the value of `$V` (which will typically be a global variable, though when the pattern is used in contexts such as the [xsl:number](#) or [xsl:for-each-group](#) instructions, it can also be a local variable).

The [selection pattern](#) `doc('product.xml')//product` matches all nodes selected by the expression `root(.)//(doc('product.xml')//product)`, that is, all `product` elements in the document whose URI is `product.xml`.

The [selection pattern](#) `root(.)/self::E` matches an `E` element that is the root of a tree (that is, an `E` element with no parent node).

Although the semantics of [selection patterns](#) are specified formally in terms of expression evaluation, it is possible to understand pattern matching using a different model. A [selection 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

A [dynamic error](#) or [type error](#) that occurs during the evaluation of a [pattern](#) against a particular item has the effect that the item being tested is treated as not matching the pattern. The error does not cause the transformation to fail, and cannot be caught by a try/catch expression surrounding the instruction that causes the pattern to be evaluated.

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.

There is a risk that ignoring errors in this way may make programming mistakes harder to debug. Implementations may mitigate this by providing warnings or other diagnostics when evaluation of a pattern triggers an error condition.

Static errors in patterns, including dynamic and type errors that are signaled statically as permitted by the specification, are reported in the normal way and cause the transformation to fail.

The requirement to detect and report a [circularity](#) as a dynamic error overrides this rule.

5.6 Value Templates

The string value of an attribute or text node in the stylesheet may in particular circumstances contain embedded expressions enclosed between curly brackets. Attributes and text nodes that use (or are permitted to use) this mechanism are referred to respectively as [attribute value templates](#) and [text value templates](#).

[**DEFINITION:** Collectively, attribute value templates and text value templates are referred to as **value templates**.]

A value template is a string consisting of an alternating sequence of fixed parts and variable parts:

- A variable part consists of an optional XPath [expression](#) enclosed in curly brackets ({{}): more specifically, a string conforming to the XPath production Expr?.

Note:

An expression within a variable part may contain an unescaped curly bracket within a [StringLiteral](#)^{XP30} or within a comment.

Currently no XPath expression starts with an opening curly bracket, so the use of {{ creates no ambiguity. If an enclosed expression ends with a closing curly bracket, no whitespace is required between this and the closing delimiter.

The fact that the expression is optional means that the string contained between the curly brackets may be zero-length, may comprise whitespace only, or may contain XPath comments. The effective value in this case is a zero-length string, which is equivalent to omitting the variable part entirely, together with its curly-bracket delimiters.

- 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 }}.

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

It is a [static error](#) if the string contained between matching curly brackets in a value template does not match the XPath production Expr?^{XP30}, 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 a value template.

[**DEFINITION:** The result of evaluating a value template is referred to as its **effective value**.] 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 as follows:
 - If an expression is present, the result of 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](#).
 - If the expression is omitted, a zero-length string.

Note:

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

In the case of an attribute value template, the effective value becomes the string value of the new attribute node. In the case of a text value template, the effective value becomes the string value of the new text node.

5.6.1 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 ({}), following the general rules for [value templates](#)].

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.

If the element containing the attribute is processed with [XSLT 1.0 behavior](#), then the rules for converting the value of the expression to a string (given in [5.6 Value Templates](#)) 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.

Note:

The above rule applies to attribute value templates but not to text value templates, since the latter were not available in XSLT 1.0.

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">
  
</xsl:template>
```

With this source

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

the result would be

```

```

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 cannot be Nested

For example:

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

is *not* allowed. Instead, use simply:

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

5.6.2 Text Value Templates

The [standard attribute](#) `[xsl:]expand-text` may appear on any element in the stylesheet, and determines whether descendant text nodes of that element are treated as text value templates. A text node in the stylesheet is treated as a text value template if (a) it is part of a [sequence constructor](#) or a child of an `xsl:text` instruction, (b) there is an ancestor element with an `[xsl:]expand-text` attribute, and (c) on the innermost ancestor element that has such an attribute, the value of the attribute is `yes`. The attribute is boolean and `MUST` therefore take one of the values `yes` (synonyms `true` or `1`) or `no` (synonyms `false` or `0`).

This section describes how text nodes are processed when the effective value is `yes`. Such text nodes are referred to as text value templates.

[**DEFINITION:** In a text node that is designated as a **text value template**, [expressions](#) can be used by surrounding each expression with curly brackets (`{}`).]

The rules for text value templates are given in [5.6 Value Templates](#). A text node whose value is a text value template results in the construction of a text node in the result of the containing sequence constructor or `xsl:text` instruction. The string value of that text node is obtained by computing the effective value of the value template.

Note:

The result of evaluating a text value template is a (possibly zero-length) text node. This text node becomes part of the result of the containing sequence constructor or `xsl:text` instruction, and is thereafter handled exactly as if the value had appeared explicitly as a text node in the stylesheet.

The way in which the effective value is computed does not depend on any `separator` attribute on a containing `xsl:value-of` or `xsl:attribute` instruction. The `separator` attribute only affects how the text node is combined with adjacent items in the result of the containing sequence constructor.

Fixed parts consisting entirely of whitespace are significant and are handled in the same way as any other fixed part. This is different from the default treatment of “boundary space” in XQuery.

Example: Using a text value template to construct message output

```
<xsl:variable name="id" select="'A123' />
<xsl:variable name="step" select="5"/>
<xsl:message expand-text="yes">Processing id={$id}, step={$step}</xsl:message>
```

This will typically output the message text Processing id=A123, step=5.

Example: Using a text value template to define the result of a function

```
<xsl:function name="f:sum" expand-text="yes" as="xs:integer">
<xsl:param name="x" as="xs:integer"/>
<xsl:param name="y" as="xs:integer"/>
  {$x + $y}
</xsl:function>
```

Note that although this is a very readable way of expressing the computation performed by the function, the semantics are somewhat complex, and this could mean that execution is inefficient. The function computes the value of $\$x + \y as an integer, and then constructs a text node containing the string representation of this integer (preceded and followed by whitespace). Because the declared result type of the function is `xs:integer`, this text node is then atomized, giving an `xs:untypedAtomic` value, and the `xs:untypedAtomic` value is then cast to an `xs:integer`.

Note:

The main motivations for adding text value templates to the XSLT language are firstly, to make it easier to construct parameterized text in contexts such as `xsl:value-of` and `xsl:message`, and secondly, to allow use of complex multi-line XPath expressions where maintaining correct indentation is important for readability. The fact that XML processors are required to normalize whitespace in attribute values means that writing such expressions within a `select` attribute is not ideal.

The facility is only present if enabled using the `[xsl:]expand-text` attribute. This is partly for backwards compatibility, and partly to avoid creating difficulties when constructing content that is rich in curly brackets, for example JavaScript code or CSS style sheets.

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, atomic values, and function items. 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:

1. A *Text node* appearing in the [stylesheet](#) (if it has not been removed in the process of whitespace stripping: see [4.3 Stripping Whitespace from the Stylesheet](#)) is processed as follows:
 - a. if the effective value of the standard attribute `[xsl:]expand-text` is `no`, or in the absence of this attribute, the text node in the stylesheet is copied to create a new parentless text node in the result of the sequence constructor.
 - b. Otherwise (the effective value of `[xsl:]expand-text` is `yes`), the text node in the stylesheet is processed as described in [5.6.2 Text Value Templates](#).
2. A [literal result element](#) is evaluated to create a new parentless element node, having the same [expanded QName](#) as the literal result element: see [11.1 Literal Result Elements](#).
3. An XSLT [instruction](#) produces a sequence of zero, one, or more items as its result. For most XSLT instructions, these items are nodes, but some instructions (such as [xsl:sequence](#) and [xsl:copy-of](#)) can also produce atomic values or function items. 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, function items, or existing nodes.
4. An [extension instruction](#) (see [24.2 Extension Instructions](#)) also produces a sequence of items as its result.

[**DEFINITION:** The result of evaluating a [sequence constructor](#) is the sequence of items formed by concatenating the results of evaluating each of the nodes in the sequence constructor, retaining order. This is referred to as the **immediate result** of the sequence constructor.]

However:

- For the effect of the [xsl:fallback](#) instruction, see [24.2.3 Fallback](#).
- For the effect of the [xsl:on-empty](#) and [xsl:on-non-empty](#) instructions, see [8.4 Conditional Content Construction](#).

The way that [immediate result](#) of a [sequence constructor](#) is used depends on the containing element in the stylesheet, and is specified in the rules for that element. It is typically one of the following:

- The [immediate result](#) may be bound to a [variable](#) or delivered as the result of a [stylesheet function](#). In this case the `as` attribute of the containing [xsl:variable](#) or [xsl:function](#) element may be used to declare its required type, and the [immediate result](#) is then converted to the required type by applying the [function conversion rules](#).

Note:

- In the absence of an `as` attribute, the result of a function is the [immediate result](#) of the sequence constructor; but the value of a variable (for backwards compatibility reasons) is a document node whose content is formed by applying the rules in [5.7.1 Constructing Complex Content](#) to the [immediate result](#).
- The function conversion rules do not merge adjacent text nodes or insert separators between adjacent items. This means it is often inappropriate to use [`xsl:value-of`](#) in the body of [`xsl:variable`](#) or [`xsl:function`](#), especially when the intent is to return an atomic result. The [`xsl:sequence`](#) instruction is designed for this purpose, and is usually a better choice.
- The result of a function, or the value of a variable, may contain nodes (such as elements, attributes, and text nodes) that are not attached to any 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, for example, when the element containing the sequence constructor is [`xsl:break`](#), [`xsl:catch`](#), [`xsl:fallback`](#), [`xsl:for-each`](#), [`xsl:for-each-group`](#), [`xsl:fork`](#), [`xsl:if`](#), [`xsl:iterate`](#), [`xsl:matching-substring`](#), [`xsl:non-matching-substring`](#), [`xsl:on-completion`](#), [`xsl:otherwise`](#), [`xsl:perform-sort`](#), [`xsl:sequence`](#), [`xsl:try`](#), 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`](#), [`xsl:assert`](#), 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](#).

[5.7.1 Constructing Complex Content](#)

Many instructions, for example [`xsl:copy`](#), [`xsl:element`](#), [`xsl:document`](#), [`xsl:result-document`](#), and [literal result elements](#), create a new parent node, and evaluate a [sequence constructor](#) forming the content of the instruction to create the attributes, namespaces, and children of the new parent node. The [immediate result](#) of the sequence constructor is processed to create the content of the new parent node as described in this section.

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 [immediate result](#) of the [sequence constructor](#) 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 [immediate result](#) of the [sequence constructor](#).
2. Any array item in the sequence (see [27.7.1 Arrays](#)) is replaced by its members, recursively. This is equivalent to applying the [array:flatten](#)^{FO31} function defined in [\[Functions and Operators 3.1\]](#).

Note:

This situation only arises if the [XPath 3.1 Feature](#) is implemented. Note that if the array contains nodes, this operation leaves the nodes in the sequence: they are not [atomized](#).

3. 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.

4. Any consecutive sequence of strings in the 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.
5. Any document node within the sequence is replaced by a sequence containing each of its children, in document order.
6. Zero-length text nodes within the sequence are removed.
7. Adjacent text nodes within the sequence are merged into a single text node.
8. Invalid items in the sequence are detected as follows.

[ERR XTDE0410] It is a [dynamic error](#) if the 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 [dynamic error](#) if the sequence used to construct the content of a document node contains a namespace node or attribute node.

[ERR XTDE0430] It is a [dynamic error](#) if the 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 [dynamic error](#) if the 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 XTDE0450] It is a [dynamic error](#) if the result sequence contains a function item.

9. If the 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 [implementation-dependent](#), it is not significant which of the duplicates is retained.

10. If an attribute *A* in the sequence has the same name as another attribute *B* that appears later in the sequence, then attribute *A* is discarded from the sequence. Before discarding attribute *A*, the processor MAY signal any [type errors](#) that would be signaled if attribute *B* were not present.
11. 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.

Except for the handling of base URI, the copying of a node follows the rules of the [`xsl:copy-of`](#) instruction with attributes `copy-namespaces="yes"` `copy-accumulators="no"` `validation="preserve"`.

Note:

This has the consequence that the type annotation and the values of the `nilled`, `is-id`, and `is-idrefs` properties are retained. However, if the node under construction (the new parent of the node being copied) uses a validation mode other than `preserve`, this will be transient: the values will be recomputed when the new parent node is validated.

12. 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](#).
13. 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 instructions [xsl:attribute](#), [xsl:comment](#), [xsl:processing-instruction](#), [xsl:namespace](#), and [xsl:value-of](#) all 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 a [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](#) (which may cause a dynamic error).
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 depends on the containing instruction; except where otherwise specified, it is a single space.

In the case of [xsl:attribute](#) and [xsl:value-of](#), the default separator is a single space when the `select` attribute is used, or a zero-length string otherwise; a different separator can be specified using the `separator` attribute of the instruction.

In the case of [xsl:comment](#), [xsl:processing-instruction](#), and [xsl:namespace](#), and when expanding a [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>
  <xsl:attribute name="g" expand-text="yes">{1 to 5}</xsl:attribute>
</doc>
```

This produces the output:

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

The difference between the three cases is as follows. 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. For the `g` attribute, the [text value template](#) constructs a text node using the rules for constructing simple content, which insert space separators between atomic values; the text node is then atomized to form the value of the attribute.

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. A `separator` on the third [`xsl:attribute`](#) instruction would also have no effect, because text value templates are evaluated without regard to the containing instruction.

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 [\[XDM 3.0\]](#) 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 `xmllns` or the string value `http://www.w3.org/2000/xmllns/`.

[**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.

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 match selection](#), documents loaded using the [document](#), [doc](#)^{FO30} or [collection](#)^{FO30} 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](#)^{FO30} or [collection](#)^{FO30} 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 [\[XDM 3.0\]](#), including the constraints imposed by the namespace fixup process. The effect of supplying a pseudo-document that does not meet these constraints is [implementation-dependent](#).

In an Infoset (see [\[XML Information Set\]](#)) created from a document conforming to [\[Namespaces in XML\]](#), 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 3.0 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 `xmllns=""`. 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, `xmllns : 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` datatype 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 [\[Namespaces in XML\]](#) and [\[Namespaces in XML 1.1\]](#). Those specifications deprecate the use of relative URI references as namespace URIs.

The rules for collation URIs are given in [\[Functions and Operators 3.0\]](#).

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 reference 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.

The base URI of an element node in the stylesheet is determined as defined in [Section 5.2 base-uri Accessor](#)^{DM30}. Some implementations may allow the output of the static analysis phase of stylesheet processing (a “compiled stylesheet”) to be evaluated in a different location from that where static analysis took place. Furthermore, stylesheet authors may in such cases wish to avoid exposing the location of resources that are private to the development environment. If the base URI of an element in the stylesheet is defined by an absolute URI appearing in an `xml:base` attribute within the stylesheet, this value **MUST** be used as the static base URI. In other cases where processing depends on the static base URI of a stylesheet module, implementations **MAY** use different values for the static base URI during static analysis and during dynamic evaluation (for example, an implementation **MAY** use different base URIs for resolving [xsl:import](#) module references and for resolving a relative reference used as an argument to the [doc](#)^{FO30} function). In such cases an implementation **MUST** document how the static base URI is computed for each situation in which it is required.

6 Template Rules

Template rules define the processing that can be applied to items that match a particular [pattern](#).

6.1 Defining Templates

```
<!-- Category: declaration -->
<xsl:template
  match? = pattern
  name? = eqname
  priority? = decimal
  mode? = tokens
  as? = sequence-type
  visibility? = "public" | "private" | "final" | "abstract" >
  <!-- Content: (xsl:context-item?, xsl:param*, sequence-constructor) -->
</xsl:template>
```

[**DEFINITION:** An [xsl:template](#) declaration defines a **template**, which contains a [sequence constructor](#); this sequence constructor is evaluated to determine the result of the template. A template can serve either as a [template rule](#), invoked by matching items 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. An

[xsl:template](#) element that has no `name` attribute MUST have no `visibility` 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](#)).

For details of the optional [xsl:context-item](#) child element, see [10.1.1 Declaring the Context Item for a Template](#).

If an `as` attribute of the [xsl:template](#) element 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.

If the `visibility` attribute is present with the value `abstract` then (a) the [sequence constructor](#) defining the template body MUST be empty: that is, the only permitted children are [xsl:context-item](#) and [xsl:param](#), and (b) there MUST be no `match` attribute.

If the parent of the [xsl:template](#) element is an [xsl:override](#) element, then either or both of the following conditions must be true:

1. There is a `name` attribute, and the [package](#) identified by the containing [xsl:use-package](#) element contains among its [components](#) a [named template](#) whose [symbolic identifier](#) is the same as that of this named template, and which has a [compatible](#) signature.
2. Both the following conditions are true:
 - a. There is a `match` attribute.
 - b. The value of the `mode` attribute, or in its absence the string `#default`, is a whitespace-separated sequence of tokens in which each token satisfies one of the following conditions:
 - i. The token is an EQName representing the name of a mode that is exposed, with visibility equal to `public`, by the package identified by the containing [xsl:use-package](#) element.
 - ii. The token is `#default`, and there is an ancestor-or-self element with a `default-mode` attribute whose value is an EQName representing the name of a mode that is exposed, with visibility equal to `public`, by the package identified by the containing [xsl:use-package](#) element.

Note:

The token `#unnamed` is not allowed because the unnamed mode never has public visibility. The token `#all` is not allowed because its intended meaning would not be obvious.

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 items 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 item used as the [context item](#).

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 an item 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 item, 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 items (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 item in the input sequence is processed by finding a [template rule](#) whose [pattern](#) matches that item. If there is more than one such template rule, 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 item, a built-in template rule is used (see [6.7 Built-in Template Rules](#)). The chosen template rule is evaluated. The rule that matches the *N*th item in the sorted sequence is evaluated with that item 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 item in the sorted sequence) are then concatenated, to form a single

sequence. They are concatenated retaining the order of the items 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 the 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 items selected by an expression instead of processing all children. The value of the `select` attribute is an [expression](#).

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 Nodes 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 [6.8 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"`

Example: Applying Templates to Atomic Values

This example reads a non-XML text file and processes it line-by-line, applying different template rules based on the content of each line:

```
<xsl:template name="main">
  <xsl:apply-templates select="unparsed-text-lines('input.txt')"/>
</xsl:template>

<xsl:template match=". [starts-with(., '==')] ">
  <h2><xsl:value-of select="replace(., '==', '')" /></h2>
</xsl:template>

<xsl:template match=". [starts-with(., '::')] ">
  <p class="indent"><xsl:value-of select="replace(., '::', '')" /></p>
</xsl:template>

<xsl:template match=". ">
  <p class="body"><xsl:value-of select=". " /></p>
</xsl:template>
```

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 selected item to match more than one [template rule](#) with a given [mode](#) M . When this happens, only one template rule is evaluated for the item. 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.

[**DEFINITION:** The **priority** of a template rule is specified by the `priority` attribute on the `xsl:template` declaration. If no priority is specified explicitly for a template rule, its [default priority](#) is used, as defined in [6.5 Default Priority for Template Rules](#).]

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

3. If this leaves more than one matching template rule, then:

- a. If the [mode](#) M has an `xsl:mode` declaration, and the attribute value `on-multiple-match="fail"` is specified in the mode declaration, a dynamic error is signaled. The error is treated as occurring in the `xsl:apply-templates` instruction, and can be recovered by wrapping that instruction in an `xsl:try` instruction.

[ERR XTDE0540] It is a [dynamic error](#) if the conflict resolution algorithm for template rules leaves more than one matching template rule when the declaration of the relevant [mode](#) has an `on-multiple-match` attribute with the value `fail`.

- b. Otherwise, of the matching template rules that remain, the one that occurs last in [declaration order](#) is used.

Note:

This was a recoverable error in XSLT 2.0, meaning that it was implementation-defined whether the error was signaled, or whether the ambiguity was resolved by taking the last matching rule in declaration order. In XSLT 3.0 this situation is not an error unless the attribute value `on-multiple-match="fail"` is specified in the mode declaration. It is also possible to request warnings when this condition arises, by means of the attribute `warning-on-multiple-match="yes"`.

6.5 Default Priority for Template Rules

[**DEFINITION:** If no `priority` attribute is specified on an `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.

1. If the top-level pattern is a `ParenthesizedExprP` then the outer parentheses are effectively stripped; these rules are applied recursively to the `UnionExprP` contained in the `ParenthesizedExprP`.
2. If the top-level pattern is a `UnionExprP` consisting of multiple alternatives separated by `|` or `union`, then the template rule is treated equivalently to a set of template rules, one for each alternative. These template rules are adjacent to each other in declaration order, and the declaration order within this set of template rules (which affects the result of `xsl:next-match` if the alternatives have the same default priority) is the order of alternatives in the `UnionExprP`.

Note:

The splitting of a template rule into multiple rules occurs only if there is no explicit `priority` attribute.

3. If the top-level pattern is an `IntersectExceptExprP` containing two or more `PathExprP` operands separated by `intersect` or `except` operators, then the priority of the pattern is that of the first `PathExprP`.
4. If the pattern is a `PredicatePattern` then its priority is 1 (one), unless the `PredicateListXP30` is empty, in which case the priority is -1 (minus one).
5. If the pattern is a `PathExprP` taking the form `/`, then the priority is -0.5 (minus 0.5).
6. If the pattern is a `PathExprP` taking the form of an `EQName` optionally preceded by a `ForwardAxisP` or has the form `processing-instruction(StringLiteralXP30)` or `processing-instruction(NCNameNames)` optionally preceded by a `ForwardAxisP`, then the priority is 0 (zero).
7. If the pattern is a `PathExprP` taking the form of an `ElementTestXP30` or `AttributeTestXP30`, optionally preceded by a `ForwardAxisP`, 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.

Default Priority of Patterns

Format	Priority	Notes
<code>element()</code>	-0.5	(equivalent to *)
<code>element(*)</code>	-0.5	(equivalent to *)

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

8. If the pattern is a [PathExprP](#) taking the form of a [DocumentTest^{XP30}](#), then if it includes no [ElementTest^{XP30}](#) or [SchemaElementTest^{XP30}](#) the priority is -0.5. If it does include an [ElementTest^{XP30}](#) or [SchemaElementTest^{XP30}](#), then the priority is the same as the priority of that [ElementTest^{XP30}](#) or [SchemaElementTest^{XP30}](#), computed according to the table above.
9. If the pattern is a [PathExprP](#) taking the form of an [`NCNameNames : * or * : NCNameNames`](#), optionally preceded by a [ForwardAxisP](#), then the priority is -0.25.
10. If the pattern is a [PathExprP](#) taking the form of any other [NodeTest^{XP30}](#), optionally preceded by a [ForwardAxisP](#), then the priority is -0.5.
11. In all other cases, 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.

6.6 Modes

[**DEFINITION:** A **mode** is a set of template rules; when the [xsl:apply-templates](#) instruction selects a set of items for processing, it identifies the rules to be used for processing those items by nominating a mode, explicitly or implicitly.] Modes allow a node in a [source tree](#) (for example) 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 [20.1 fn:document](#)).

Modes are identified by an [expanded QName](#); in addition to any named modes, there is always one unnamed mode available. Whether a mode is named or unnamed, its properties **MAY** be defined in an [xsl:mode](#) declaration. If a mode name is used (for example in an [xsl:template](#) declaration or an [xsl:apply-templates](#) instruction) and no declaration of that mode appears in the stylesheet, the mode is implicitly declared with default properties.

6.6.1 Declaring Modes

```
<!-- Category: declaration -->
<xsl:mode
  name? = eqname
  streamable? = boolean
  use-accumulators? = tokens
  on-no-match? = "deep-copy" | "shallow-copy" | "deep-skip" | "shallow-skip" |
  "text-only-copy" | "fail"
  on-multiple-match? = "use-last" | "fail"
  warning-on-no-match? = boolean
  warning-on-multiple-match? = boolean
  typed? = boolean | "strict" | "lax" | "unspecified"
  visibility? = "public" | "private" | "final" />
```

[**DEFINITION:** The **unnamed mode** is the default mode used when no **mode** attribute is specified on an [xsl:apply-templates](#) instruction or [xsl:template](#) declaration, unless a different default mode has been specified using the [\[xsl:\]default-mode](#) attribute of a containing element.]

Every [mode](#) other than the [unnamed mode](#) is identified by an [expanded QName](#).

A [stylesheet](#) may contain multiple [xsl:mode](#) declarations and may include or import [stylesheet modules](#) that also contain [xsl:mode](#) declarations. The name of an [xsl:mode](#) declaration is the value of its **name** attribute, if any.

[**DEFINITION:** All the [xsl:mode](#) declarations in a [package](#) that share the same name are grouped into a named **mode definition**; those that have no name are grouped into a single unnamed mode definition.]

The **declared-modes** attribute of the [xsl:package](#) element determines whether implicit mode declarations are allowed, as described in [3.5.4.1 Requiring Explicit Mode Declarations](#). If the package allows implicit mode declarations, then if a [stylesheet](#) does not contain a declaration of the unnamed mode, a declaration is implied equivalent to an [xsl:mode](#) element with no attributes. Similarly, if there is a mode that is named in an [xsl:template](#) or [xsl:apply-templates](#) element, or in the [\[xsl:\]default-mode](#) attribute of a containing element, and the [stylesheet](#) does not contain a declaration of that mode, then a declaration is implied comprising an [xsl:mode](#) element with a **name** attribute equal to that mode name, plus the attribute **visibility="private"**.

The attributes of the `xsl:mode` declaration establish values for a number of properties of a mode. The allowed values and meanings of the attributes are given in the following table.

Attributes of the `xsl:mode` Element

Attribute	Values	Meaning
name	An EQName	Specifies the name of the mode. If omitted, this <code>xsl:mode</code> declaration provides properties of the unnamed mode
streamable	yes or no (default no)	Determines whether template rules in this mode are to be capable of being processed using streaming . If the value yes is specified, then the body of any template rule that uses this mode MUST conform to the rules for streamable templates given in 6.6.4 Streamable Templates .
use-accumulators	List of accumulator names, or #all (default is an empty list)	Relevant only when this mode is the initial mode of the transformation, determines which accumulators are applicable to documents containing nodes in the initial match selection . For further details see 18.2.2 Applicability of Accumulators .
on-no-match	One of deep-copy, shallow-copy, deep-skip, shallow-skip, text-only-copy or fail (default text-only-copy)	Determines selection of the built-in template rules that are used to process an item when an <code>xsl:apply-templates</code> instruction selects an item that does not match any user-written template rule in the stylesheet . For details, see 6.7 Built-in Template Rules .
on-multiple-match	One of fail or use-last (default use-last)	Defines the action to be taken when <code>xsl:apply-templates</code> is used in this mode and more than one user-written template rule is available to process an item, each having the same import precedence and priority . The value fail indicates that it is a dynamic error if more than one template rule matches an item. The value use-last indicates that the situation is not to be treated as an error (the last template in declaration order is the one that is used).
warning-on-no-match	One of yes or no. The default is implementation-defined	Requests the processor to output (or not to output) a warning message in the case where an <code>xsl:apply-templates</code> instruction selects an item that matches no user-written template rule. The form and destination of such warnings is implementation-defined . The processor MAY ignore this attribute, for example if the environment provides no suitable means of communicating with the user.

Attribute	Values	Meaning
warning-on-multiple-match	One of yes or no. The default is implementation-defined	Requests the processor to output a warning message in the case where an xsl:apply-templates instruction selects an item that matches multiple template rules having the same import precedence and priority . The form and destination of such warnings is implementation-defined . The processor MAY ignore this attribute, for example if the environment provides no suitable means of communicating with the user.
typed	One of yes, no, strict , lax , or unspecified . The default is unspecified .	See 6.6.3 Declaring the Type of Nodes Processed by a Mode .
visibility	One of public , private , or final . The default is private .	See 3.5.3.1 Visibility of Components . If the mode is unnamed, that is, if the name attribute is absent, then the visibility attribute if present MUST have the value private . A named mode is not eligible to be used as the initial mode if its visibility is private .

[DEFINITION: A **streamable mode** is a [mode](#) that is declared in an [xsl:mode](#) declaration with the attribute `streamable="yes"`.]

For any named [mode](#), the effective value of each attribute is taken from an [xsl:mode](#) declaration that has a matching name in its [name](#) attribute, 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 the [unnamed mode](#), the effective value of each attribute is taken from an [xsl:mode](#) declaration that has no [name](#) attribute, 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 XTSE0545] It is a [static error](#) if for any named or unnamed [mode](#), a package explicitly specifies two conflicting values for the same attribute in different [xsl:mode](#) declarations having the same [import precedence](#), unless there is another definition of the same attribute with higher import precedence. The attributes in question are the attributes other than [name](#) on the [xsl:mode](#) element.

6.6.2 Using Modes

[DEFINITION: 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 specified in the [\[xsl:\]default-mode](#) attribute of the innermost containing element that has such an attribute, which in turn defaults to the [unnamed mode](#). If the [mode](#) 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 in the mode attribute **MUST** be one of the following:

- an [EQName](#), which is expanded as described in [5.1.1 Qualified Names](#) to define the name of the mode
- the token #default, to indicate that the template rule is applicable to the default mode that would apply if the mode attribute were absent
- the token #unnamed, to indicate that the template rule is applicable to the [unnamed mode](#)
- the token #all, to indicate that the template rule is applicable to all modes (specifically, to the unnamed mode and to every mode that is named explicitly or implicitly in an [xsl:apply-templates](#) instruction anywhere in the stylesheet).

When a template rule specifies mode="#all" this is interpreted as meaning all modes declared implicitly or explicitly within the [declaring package](#) of the [xsl:template](#) element. This value cannot be used in the case of a template rule declared within an [xsl:override](#) element.

[ERR XTSE0550] It is a [static error](#) if the list of modes 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 XTSE3440] In the case of a [template rule](#) (that is, an [xsl:template](#) element having a [match](#) attribute) appearing as a child of [xsl:override](#), it is a [static error](#) if the list of modes in the mode attribute contains #all or #unnamed, or if it contains #default and the default mode is the [unnamed mode](#), or if the mode attribute is omitted when the default mode is the [unnamed mode](#).

The [xsl:apply-templates](#) element also has an optional mode attribute. The value of this attribute **MUST** be one of the following:

- an [EQName](#), which is expanded as described in [5.1.1 Qualified Names](#) to define the name of a mode
- the token #default, to indicate that the default mode for the [stylesheet module](#) is to be used
- the token #unnamed, to indicate that the [unnamed mode](#) is to be used
- the token #current, to indicate that the [current mode](#) is to be used

If the attribute is omitted, the default mode for the [stylesheet module](#) is used.

When searching for a template rule to process each item 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 [initial mode](#), 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 [non-contextual function call](#) is made, the current mode is set to the [unnamed mode](#). While evaluating global variables and parameters, and the sequence constructor contained in [xsl:key](#) or [xsl:sort](#), the current mode is set to the unnamed mode. No other instruction changes the current mode. The current mode while evaluating an [attribute set](#) is the same as the current mode of the caller. 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 [6.8 Overriding Template Rules](#)).

[6.6.3 Declaring the Type of Nodes Processed by a Mode](#)

Typically the template rules in a particular `mode` will be designed to process a specific kind of input document. The `typed` attribute of `xsl:mode` gives the stylesheet author the opportunity to provide information about this document to the processor. This information may enable the processor to improve diagnostics or to optimize performance.

The `typed` attribute of `xsl:mode` informs the processor whether the nodes to be processed by template rules in this mode are to be typed or untyped.

- If the value `yes` is specified (synonyms `true` or `1`), then all nodes processed in this mode must be typed. A dynamic error occurs if `xsl:apply-templates` in this mode selects an element or attribute node whose `type annotation` is `xs:untyped` or `xs:untypedAtomic`.
- If the value `no` is specified (synonyms `false` or `0`), then all nodes processed in this mode must be untyped. A dynamic error occurs if `xsl:apply-templates` in this mode selects an element or attribute whose `type annotation` is anything other than `xs:untyped` or `xs:untypedAtomic`.
- The value `strict` is equivalent to `yes`, with the additional provision that in the match pattern of any template rule that is `applicable` to this mode, any `NameTest` used in the `ForwardStepP` of the first `StepExprP` of a `RelativePathExprP` is interpreted as follows:
 - If the `NameTest` is an `EQName E`, and the principal node kind of the axis of this step is `Element`, then:
 - It is a static error if the in-scope schema declarations do not include a global element declaration for element name `E`
 - When matching templates in this mode, the element name `E` appearing in this step is interpreted as `schema-element(E)`. (Informally, this means that it will only match an element if it has been validated against this element declaration).
 - Otherwise (the `NameTest` is a wildcard or the principal node kind is `Attribute` or `Namespace`), the template matching proceeds as if the `typed` attribute were absent.
- The value `lax` is equivalent to `yes`, with the additional provision that in the match pattern of any template rule that is `applicable` to this mode, any `NameTest` used in the `ForwardStepP` of the first `StepExprP` of a `RelativePathExprP` is interpreted as follows:
 - If the `NameTest` is an `EQName E`, and the principal node kind of the axis of this step is `Element`, and the in-scope schema declarations include a global element declaration for element name `E`, then:
 - When matching templates in this mode, the element name `E` appearing in this step is interpreted as `schema-element(E)`. (Informally, this means that it will only match an element if it has been validated against this element declaration).
 - Otherwise (the `NameTest` is a wildcard, or the principal node kind is `Attribute` or `Namespace`, or there is no element declaration for `E`), the template matching proceeds as if the `typed` attribute were absent.

[ERR XTTE3100] It is a `type error` if an `xsl:apply-templates` instruction in a particular `mode` selects an element or attribute whose type is `xs:untyped` or `xs:untypedAtomic` when the `typed` attribute of that mode specifies the value `yes`, `strict`, or `lax`.

[ERR XTSE3105] It is a `static error` if a template rule applicable to a mode that is defined with `typed="strict"` uses a match pattern that contains a `RelativePathExprP` whose first `StepExprP` is an `AxisStepP` whose `ForwardStepP` uses an axis whose principal node kind is `Element` and whose `NodeTest` is an `EQName` that does not correspond to the name of any global element declaration in the `in-scope schema components`.

[ERR_XTTE3110] It is a [type error](#) if an `xsl:apply-templates` instruction in a particular mode selects an element or attribute whose type is anything other than `xs:untyped` or `xs:untypedAtomic` when the `typed` attribute of that mode specifies the value no.

6.6.4 Streamable Templates

A template rule that is [applicable](#) to a mode M is [guaranteed-streamable](#) if and only if all the following conditions are satisfied:

1. Mode M is declared in an `xsl:mode` declaration that specifies `streamable="yes"`.
2. The [pattern](#) defined in the `match` attribute of the `xsl:template` element is a [motionless](#) pattern as defined in [19.8.10 Classifying Patterns](#).
3. The [sweep](#) of the [sequence constructor](#) forming the body of the `xsl:template` element is either [motionless](#) or [consuming](#).
4. The [type-adjusted posture](#) of the [sequence constructor](#) forming the body of the `xsl:template` element, with respect to the [U-type](#) that corresponds to the declared return type of the template (defaulting to `item(*)`), is [grounded](#).

Note:

This means that either (a) the sequence constructor is grounded as written (that is, it does not return streamed nodes), or (b) it effectively becomes grounded because the declared result type of the template is atomic, leading to implicit atomization of the result.

5. Every [expression](#) and contained [sequence constructor](#) in a contained `xsl:param` element (the construct that provides the default value of the parameter) is [motionless](#).

Specifying `streamable="yes"` on an `xsl:mode` declaration declares an intent that every template rule that includes that mode (explicitly or implicitly, including by specifying `#all`), should be streamable, either because it is [guaranteed-streamable](#), or because it takes advantage of streamability extensions offered by a particular processor. The consequences of declaring the mode to be streamable when there is such a template rule that is not guaranteed streamable depend on the conformance level of the processor, and are explained in [19.10 Streamability Guarantees](#).

Processing of a document using streamable templates may be initiated using code such as the following, where S is a mode declared with `streamable="yes"`:

```
<xsl:source-document streamable="yes" href="bigdoc.xml">
  <xsl:apply-templates mode="S"/>
</xsl:source-document>
```

Alternatively, streamed processing may be initiated by invoking the transformation with an [initial mode](#) declared as streamable, while supplying the [initial match selection](#) (in an [implementation-defined](#) way) as a streamed document.

Note:

Invoking a streamable template using the construct `<xsl:apply-templates select="doc('bigdoc.xml')"/>` does not ensure streamed processing. As always, processors may use streamed processing if they are able to do so, but when the `doc`^{FO30} or `document` functions are used, processors are obliged to ensure that the results are deterministic, which may be difficult to reconcile with streaming (if the same document is read twice, the results must be identical). The use of `xsl:source-document` with `streamable="yes"` does not offer the same guarantees of determinism.

For an example of processing a collection of documents by use of the function `uri-collection`^{FO30} in conjunction with `xsl:source-document`, see [18.1.2 Examples of xsl:source-document](#).

6.7 Built-in Template Rules

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

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.

There are six sets of built-in template rules available. The set that is chosen is a property of the `mode` selected by the `xsl:apply-templates` instruction. This property is set using the `on-no-match` attribute of the `xsl:mode` declaration, which takes one of the six values `deep-copy`, `shallow-copy`, `deep-skip`, `shallow-skip`, `text-only-copy`, or `fail`, the default being `text-only-copy`. The effect of these six sets of built-in template rules is explained in the following subsections.

6.7.1 Built-in Templates: Text-only Copy

The effect of processing a tree using a `mode` that specifies `on-no-match="text-only-copy"` is that the textual content of the source document is retained while losing the markup, except where explicit template rules dictate otherwise. When an element is encountered for which there is no explicit `template rule`, the processing continues with the children of that element. Text nodes are copied to the output.

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.

This is equivalent to the following in the case where there are no parameters:

```
<xsl:template match="document-node()|element()" mode="M">
  <xsl:apply-templates mode="#current"/>
</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="M">
  <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 atomic values returns a text node containing the value. It is effectively:

```
<xsl:template match=". [. instance of xs:anyAtomicType]" mode="M">
  <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, comments, and namespace nodes does nothing (it returns the empty sequence).

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

The built-in [template rule](#) for functions (including maps) does nothing (it returns the empty sequence).

```
<xsl:template
  match=". [. instance of function(*)]"
  mode="M"/>
```

The built-in [template rule](#) for arrays (see [27.7.1 Arrays](#)) is to apply templates to the members of the array. It is equivalent to invoking [xsl:apply-templates](#) with the `select` attribute set to `?*` (which selects the members of the array), 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.

This is equivalent to the following in the case where there are no parameters:

```
<xsl:template match=". [. instance of array(*)]" mode="M">
  <xsl:apply-templates mode="#current" select="?*"/>
</xsl:template>
```

The following example illustrates the use of built-in template rules when there are parameters.

Example: Using a Built-In Template Rule

Suppose the stylesheet contains the following instruction:

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

6.7.2 Built-in Templates: Deep Copy

The effect of processing a tree using a `mode` that specifies `on-no-match="deep-copy"` is that an unmatched element in the source tree is copied unchanged to the output, together with its entire subtree. Other unmatched items are also copied unchanged. The subtree is copied unconditionally, without attempting to match nodes in the subtree against template rules.

When this default action is selected for a mode M , all items (nodes, atomic values, and functions, including maps and arrays) are processed using a template rule that is equivalent to the following:

```
<xsl:template match=". " mode="M">
  <xsl:copy-of select=". " validation="preserve"/>
</xsl:template>
```

6.7.3 Built-in Templates: Shallow Copy

The effect of processing a tree using a `mode` that specifies `on-no-match="shallow-copy"` is that the source tree is copied unchanged to the output, except for nodes where different processing is specified using an explicit [template rule](#).

When this default action is selected for a mode M , all items (nodes, atomic values, and functions, including maps and arrays) are processed using a template rule that is equivalent to the following, except that all parameters supplied in `xsl:with-param` elements are passed on implicitly to the called templates:

```
<xsl:template match=". " mode="M">
  <xsl:copy validation="preserve">
    <xsl:apply-templates select="@*" mode="M"/>
    <xsl:apply-templates select="node()" mode="M"/>
  </xsl:copy>
</xsl:template>
```

This rule is often referred to as the *identity template*, though it should be noted that it does not preserve node identity.

Note:

This rule differs from the traditional identity template rule by using two [xsl:apply-templates](#) instructions, one to process the attributes and one to process the children. The only observable difference from the traditional `select="node() | @*"` is that with two separate instructions, the value of `position()` in the called templates forms one sequence starting at 1 for the attributes, and a new sequence starting at 1 for the children.

Example: Modified Identity Transformation

The following stylesheet transforms an input document by deleting all elements named `note`, together with their attributes and descendants:

```
<xsl:stylesheet version="3.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    <xsl:mode on-no-match="shallow-copy" streamable="true"/>

    <xsl:template match="note">
        <!-- no action -->
    </xsl:template>

</xsl:stylesheet>
```

6.7.4 [Built-in Templates: Deep Skip](#)

The effect of processing a tree using a `mode` that specifies `on-no-match="deep-skip"` is that where no explicit template rule is specified for an element, that element and all its descendants are ignored, and are not copied to the result tree.

The effect of choosing `on-no-match="deep-skip"` is as follows:

- The built-in rule for document 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.

In the case where there are no parameters, this is equivalent to the following rule:

```
<xsl:template match="document-node()" mode="M">
    <xsl:apply-templates mode="#current"/>
</xsl:template>
```

- The built-in rule for all items other than document nodes (that is, for all other kinds of node, as well as atomic values and functions, including maps and arrays) is to do nothing, that is, to return an empty sequence (without applying templates to any children or ancestors).

This is equivalent to the following rule:

```
<xsl:template match=". ." mode="M"/>
```

6.7.5 Built-in Templates: Shallow Skip

The effect of processing a tree using a [mode](#) that specifies `on-no-match="shallow-skip"` is to drop both the textual content and the markup from the result document, except where there is an explicit user-written [template rule](#) that dictates otherwise.

The built-in rule for document nodes and element nodes applies templates (in the current mode) first to the node's attributes and then to its children. If the built-in rule was invoked with parameters, those parameters are passed on in the implicit [xsl:apply-templates](#) instructions.

In the case where there are no parameters, this is equivalent to the following rule:

```
<xsl:template match="document-node()|element()" mode="M">
  <xsl:apply-templates select="@*" mode="#current"/>
  <xsl:apply-templates mode="#current"/>
</xsl:template>
```

The built-in template rule for all other kinds of node, and for atomic values and functions (including maps, but not arrays) is empty: that is, when the item is matched, the built-in template rule returns an empty sequence.

This is equivalent to the following rule:

```
<xsl:template match=". " mode="M"/>
```

The built-in [template rule](#) for arrays (see [27.7.1 Arrays](#)) is to apply templates to the members of the array. It is equivalent to invoking [xsl:apply-templates](#) with the `select` attribute set to `?*` (which selects the members of the array), 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.

This is equivalent to the following in the case where there are no parameters:

```
<xsl:template match=". [ . instance of array(*) ]" mode="M">
  <xsl:apply-templates mode="#current" select="?*"/>
</xsl:template>
```

6.7.6 Built-in Templates: Fail

The effect of choosing `on-no-match="fail"` for a [mode](#) is that every item selected in an [xsl:apply-templates](#) instruction must be matched by an explicit user-written [template rule](#).

The built-in template rule is effectively:

```
<xsl:template match=". " mode="M">
  <xsl:message terminate="yes" error-code="err:XTDE0555"/>
</xsl:template>
```

with an [implementation-dependent](#) message body.

[ERR XTDE0555] It is a [dynamic error](#) if [`xsl:apply-templates`](#), [`xsl:apply-imports`](#) or [`xsl:next-match`](#) is used to process a node using a mode whose declaration specifies `on-no-match="fail"` when there is no [template rule](#) in the [stylesheet](#) whose match pattern matches that node.

6.8 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, and also declarations of the same precedence and priority that appear earlier in [declaration order](#). Both instructions will invoke the built-in template rule for the context item (see [6.7 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.]

The [current template rule](#) is cleared (becomes [absent](#)) by any instruction that evaluates an operand with changed focus. It is therefore cleared when evaluating [instructions](#) contained within:

- [`xsl:for-each`](#)
- [`xsl:for-each-group`](#)
- [`xsl:analyze-string`](#)
- [`xsl:iterate`](#)
- [`xsl:source-document`](#)
- [`xsl:merge`](#)
- [`xsl:sort`](#)
- [`xsl:key`](#)
- [`xsl:copy`](#) if and only if there is a `select` attribute
- A global [`xsl:variable`](#) or [`xsl:param`](#)
- [`xsl:function`](#)
- [`xsl:template`](#) if and only if the called template specifies `<xsl:context-item use="absent"/>`

Note:

The current template rule is not affected by invoking named attribute sets (see [10.2 Named Attribute Sets](#)), or named templates (see [10.1 Named Templates](#)) unless `<xsl:context-item use="absent"/>` is specified. 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 [absent](#).

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.

Both [xsl:apply-imports](#) and [xsl:next-match](#) search for a [template rule](#) that matches the [context item](#), and that is applicable to the [current mode](#) (see [6.6 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.

[ERR XTSE3460] It is a [static error](#) if an [xsl:apply-imports](#) element appears in a [template rule](#) declared within an [xsl:override](#) element. (To invoke the template rule that is being overridden, [xsl:next-match](#) should therefore be used.)

- 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, plus the 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 with no [priority](#) attribute, 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 item matches more than one alternative, 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 template rule has an explicit [priority](#).

Note:

Because a template rule declared as a child of `xsl:override` has higher precedence than any template rule declared in the used package (see [3.5.4 Overriding Template Rules from a Used Package](#)), the effect of `xsl:next-match` within such a template rule is to consider as candidates first any other template rules for the same mode within the same `xsl:use-package` element (taking into account explicit and implicit priority, and document order, in the usual way), and then all template rules in the used package.

If a matching template rule R is found, then the result of the `xsl:next-match` or `xsl:apply-imports` instruction is the result of invoking R , with the values of parameters being set using the child `xsl:with-param` elements as described in [9.10 Setting Parameter Values](#). The template rule R is evaluated with the same `focus` as the `xsl:next-match` or `xsl:apply-imports` instruction. The `current template rule` changes to be R . The `current mode` does not change.

Note:

In the case where the current template rule T is declared within an `xsl:override` element in a using package P , while the selected rule R is declared within a different package Q , and where the current mode is M_P (mode M in package P), the effect is that the current mode for evaluation of R remains M_P rather than reverting to its corresponding mode M_Q (mode M in package Q). If R contains an `xsl:apply-templates` instruction that uses `mode="#current"`, then the set of template rules considered by this instruction will therefore include any overriding template rules declared in P as well as the original rules declared in Q .

If no matching template rule is found that satisfies these criteria, the built-in template rule for the context item is used (see [6.7 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 [9.10 Setting Parameter Values](#)). It also passes on any `tunnel parameters` as described in [10.1.3 Tunnel Parameters](#).

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

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 or 3.0 processor, but can be used to define fallback behavior when the stylesheet is processed by an XSLT 1.0 processor with forwards compatible behavior.

6.9 Passing Parameters to Template Rules

A template rule may have parameters. The parameters are declared in the body of the template using [xsl:param](#) elements, as described in [9.2 Parameters](#).

Values for these parameters may be supplied in the calling [xsl:apply-templates](#), [xsl:apply-imports](#), or [xsl:next-match](#) instruction by means of [xsl:with-param](#) elements appearing as children of the calling instruction. The [expanded QName](#) represented by the `name` attribute of the [xsl:with-param](#) element must match the [expanded QName](#) represented by the `name` attribute of the corresponding [xsl:param](#) element.

It is not an error for these instructions to supply a parameter that does not match any parameter declared in the template rule that is invoked; unneeded parameter values are simply ignored.

A parameter may be declared as a [tunnel parameter](#) by specifying `tunnel="yes"` in the [xsl:param](#) declaration; in this case the caller must supply the value as a tunnel parameter by specifying `tunnel="yes"` in the corresponding [xsl:with-param](#) element. Tunnel parameters differ from ordinary template parameters in that they are passed transparently through multiple template invocations. They are fully described in [10.1.3 Tunnel Parameters](#).

7 Repetition

XSLT offers two constructs for processing each item of a sequence: [xsl:for-each](#) and [xsl:iterate](#).

The main difference between the two constructs is that with [xsl:for-each](#), the processing applied to each item in the sequence is independent of the processing applied to any other item; this means that the items may be processed in any order or in parallel, though the order of the output sequence is well defined and corresponds to the order of the input (sorted if so requested). By contrast, with [xsl:iterate](#), the processing is explicitly sequential: while one item is being processed, values may be computed which are then available for use while the next item is being processed. This makes [xsl:iterate](#) suitable for tasks such as creating a running total over a sequence of financial transactions.

A further difference is that [xsl:for-each](#) permits sorting of the input sequence, while [xsl:iterate](#) does not.

7.1 The [xsl:for-each](#) instruction

```
<!-- 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; it contains an [expression](#) which is evaluated to produce 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.
- 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.

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>
```

7.2 The xsl:iterate Instruction

```
<!-- Category: instruction -->
<xsl:iterate
  select = expression >
  <!-- Content: (xsl:param*, xsl:on-completion?, sequence-constructor) -->
</xsl:iterate>
```

```
<!-- Category: instruction -->
<xsl:next-iteration>
  <!-- Content: (xsl:with-param*) -->
</xsl:next-iteration>
```

```
<!-- Category: instruction -->
<xsl:break
  select? = expression >
  <!-- Content: sequence-constructor -->
</xsl:break>
```

```
<xsl:on-completion
  select? = expression >
  <!-- Content: sequence-constructor -->
</xsl:on-completion>
```

The `select` attribute is REQUIRED; it contains an [expression](#) which is evaluated to produce a sequence, called the input sequence.

The [sequence constructor](#) contained in the [xsl:iterate](#) instruction is evaluated once for each item in the input sequence, in order, or until the loop exits by evaluating an [xsl:break](#) instruction, whichever is earlier. Within the [sequence constructor](#) that forms the body of the [xsl:iterate](#) instruction, the [context item](#) is set to each item from the value of the `select` expression in turn; the [context position](#) reflects the position of this item in the input sequence, and the [context size](#) is the number of items in the input sequence (which may be greater than the number of iterations, if the loop exits prematurely using [xsl:break](#)).

Note:

If [xsl:iterate](#) is used in conjunction with [xsl:source-document](#) to achieve streaming, calls on the function [`last`^{FO30}](#) will be disallowed.

The [xsl:break](#) and [xsl:on-completion](#) elements may have either a `select` attribute or a non-empty contained [sequence constructor](#) but not both. The effect of the element in both cases is obtained by evaluating the `select` expression if present or the contained sequence constructor otherwise; if neither is present, the value is an empty sequence.

Note:

The `xsl:on-completion` element appears before other children of `xsl:iterate` to ensure that variables declared in the sequence constructor are not in scope within `xsl:on-completion`, since such variables do not have a defined value within `xsl:on-completion` especially in the case where the value of the `select` attribute is an empty sequence.

The effect of `xsl:next-iteration` is to cause the iteration to continue by processing the next item in the input sequence, potentially with different values for the iteration parameters. The effect of `xsl:break` is to cause the iteration to finish, whether or not all the items in the input sequence have been processed. In both cases the affected iteration is the one controlled by the innermost ancestor `xsl:iterate` element.

The instructions `xsl:next-iteration` and `xsl:break` are allowed only as descendants of an `xsl:iterate` instruction, and only in a `tail position` within the `sequence constructor` forming the body of the `xsl:iterate` instruction.

[DEFINITION: An `instruction` J is in a **tail position** within a `sequence constructor` SC if it satisfies one of the following conditions:

- J is the last instruction in SC , ignoring any `xsl:fallback` instructions.
- J is in a `tail position` within the sequence constructor that forms the body of an `xsl:if` instruction that is itself in a `tail position` within SC .
- J is in a `tail position` within the sequence constructor that forms the body of an `xsl:when` or `xsl:otherwise` branch of an `xsl:choose` instruction that is itself in a `tail position` within SC .
- J is in a `tail position` within the sequence constructor that forms the body of an `xsl:try` instruction that is itself in a `tail position` within SC (that is, it is immediately followed by an `xsl:catch` element, ignoring any `xsl:fallback` elements).
- J is in a `tail position` within the sequence constructor that forms the body of an `xsl:catch` element within an `xsl:try` instruction that is itself in a `tail position` within SC .

]

[ERR XTSE3120] It is a `static error` if an `xsl:break` or `xsl:next-iteration` element appears other than in a `tail position` within the `sequence constructor` forming the body of an `xsl:iterate` instruction.

[ERR XTSE3125] It is a `static error` if the `select` attribute of `xsl:break` or `xsl:on-completion` is present and the instruction has children.

[ERR XTSE3130] It is a `static error` if the `name` attribute of an `xsl:with-param` child of an `xsl:next-iteration` element does not match the `name` attribute of an `xsl:param` child of the innermost containing `xsl:iterate` instruction.

Parameter names in `xsl:with-param` must be unique: [see [ERR XTSE0670](#)].

The result of the `xsl:iterate` instruction is the concatenation of the sequences that result from the repeated evaluation of the contained `sequence constructor`, followed by the sequence that results from evaluating the `xsl:break` or `xsl:on-completion` element if any.

Any [xsl:param](#) element that appears as a child of [xsl:iterate](#) declares a parameter whose value may vary from one iteration to the next. The initial value of the parameter is the value obtained according to the rules given in [9.3 Values of Variables and Parameters](#). The dynamic context for evaluating the initial value of an [xsl:param](#) element is the same as the dynamic context for evaluating the [select](#) expression of the [xsl:iterate](#) instruction (the context item is thus *not* the first item in the input sequence).

On the first iteration a parameter always takes its initial value (which may depend on variables or other aspects of the dynamic context). Subsequently:

- If an [xsl:next-iteration](#) instruction is evaluated, then parameter values for processing the next item in the input sequence can be set in the [xsl:with-param](#) children of that instruction; in the absence of an [xsl:with-param](#) element that names a particular parameter, that parameter will retain its value from the previous iteration.
- If an [xsl:break](#) instruction is evaluated, no further items in the input sequence are processed.
- If neither an [xsl:next-iteration](#) nor an [xsl:break](#) instruction is evaluated, then the next item in the input sequence is processed using parameter values that are unchanged from the previous iteration.

The [xsl:next-iteration](#) instruction contributes nothing to the result sequence (technically, it returns an empty sequence). The instruction supplies parameter values for the next iteration, which are evaluated according to the rules given in [9.10 Setting Parameter Values](#); if there are no further items in the input sequence then it supplies parameter values for use while evaluating the body of the [xsl:on-completion](#) element if any.

The [xsl:break](#) instruction indicates that the iteration should terminate without processing any remaining items from the input sequence. The [select](#) expression or contained sequence constructor is evaluated using the same context item, position, and size as the [xsl:break](#) instruction itself, and the result is appended to the result of the [xsl:iterate](#) instruction as a whole.

If neither an [xsl:next-iteration](#) nor an [xsl:break](#) instruction is evaluated, the next item in the input sequence is processed with parameter values unchanged from the previous iteration; if there are no further items in the input sequence, the iteration terminates.

The optional [xsl:on-completion](#) element (which is not technically an [instruction](#) and is not technically part of the [sequence constructor](#)) is evaluated when the input sequence is exhausted. It is not evaluated if the evaluation is terminated using [xsl:break](#). During evaluation of its [select](#) expression or sequence constructor the context item, position, and size are [absent](#) (that is, any reference to these values is an error). However, the values of the parameters to [xsl:iterate](#) are available, and take the values supplied by the [xsl:next-iteration](#) instruction evaluated while processing the last item in the sequence.

If the input sequence is empty, then the result of the [xsl:iterate](#) instruction is the result of evaluating the [select](#) attribute or [sequence constructor](#) forming the body of the [xsl:on-completion](#) element, using the initial values of the [xsl:param](#) elements. If there is no [xsl:on-completion](#) element, the result is an empty sequence.

Note:

Conceptually, `xsl:iterate` behaves like a tail-recursive function. The `xsl:next-iteration` instruction then represents the recursive call, supplying the tail of the input sequence as an implicit parameter. There are two main reasons for providing the `xsl:iterate` instruction. One is that many XSLT users find writing recursive functions to be a difficult skill, and this construct promises to be easier to learn. The other is that recursive function calls are difficult for an optimizer to analyze. Because `xsl:iterate` is more constrained than a general-purpose head-tail recursive function, it should be more amenable to optimization. In particular, when the instruction is used in conjunction with `xsl:source-document`, it is designed to make it easy for the implementation to use streaming techniques, processing the nodes in an input document sequentially as they are read, without building the entire document tree in memory.

The examples below use `xsl:iterate` in conjunction with the `xsl:source-document` instruction. This is not the only way of using `xsl:iterate`, but it illustrates the way in which the two features can be combined to achieve streaming of a large input document.

Example: Using `xsl:iterate` to Compute Cumulative Totals

Suppose that the input XML document has this structure

```
<transactions>
  <transaction date="2008-09-01" value="12.00"/>
  <transaction date="2008-09-01" value="8.00"/>
  <transaction date="2008-09-02" value="-2.00"/>
  <transaction date="2008-09-02" value="5.00"/>
</transactions>
```

and that the requirement is to transform this to:

```
<account>
  <balance date="2008-09-01" value="12.00"/>
  <balance date="2008-09-01" value="20.00"/>
  <balance date="2008-09-02" value="18.00"/>
  <balance date="2008-09-02" value="23.00"/>
</account>
```

This can be achieved using the following code, which is designed to process the transaction file using streaming:

```
<account>
  <xsl:source-document streamable="yes" href="transactions.xml">
    <xsl:iterate select="transactions/transaction">
      <xsl:param name="balance" select="0.00" as="xs:decimal"/>
      <xsl:variable name="newBalance"
                    select="$balance + xs:decimal(@value)"/>
      <balance date="{@date}" value="{format-number($newBalance, '0.00')}"/>
      <xsl:next-iteration>
        <xsl:with-param name="balance" select="$newBalance"/>
      </xsl:next-iteration>
    </xsl:iterate>
  </xsl:source-document>
</account>
```

The following example modifies this by only outputting the information for the first day's transactions:

```

<account>
  <xsl:source-document streamable="yes" href="transactions.xml">
    <xsl:iterate select="transactions/transaction">
      <xsl:param name="balance" select="0.00" as="xs:decimal"/>
      <xsl:param name="prevDate" select="()" as="xs:date?"/>
      <xsl:variable name="newBalance"
                    select="$balance + xs:decimal(@value)"/>
      <xsl:variable name="thisDate"
                    select="xs:date(@date)"/>
      <xsl:choose>
        <xsl:when test="empty($prevDate) or $thisDate eq $prevDate">
          <balance date="${thisDate}"
                    value="{format-number($newBalance, '0.00')}"/>
          <xsl:next-iteration>
            <xsl:with-param name="balance" select="$newBalance"/>
            <xsl:with-param name="prevDate" select="$thisDate"/>
          </xsl:next-iteration>
        </xsl:when>
        <xsl:otherwise>
          <xsl:break/>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:iterate>
  </xsl:source-document>
</account>

```

The following code outputs the balance only at the end of each day, together with the final balance:

```

<account>
  <xsl:source-document streamable="yes" href="transactions.xml">
    <xsl:iterate select="transactions/transaction">
      <xsl:param name="balance" select="0.00" as="xs:decimal"/>
      <xsl:param name="prevDate" select="()" as="xs:date?"/>
      <xsl:on-completion>
        <balance date="${prevDate}"
                  value="{format-number($balance, '0.00')}"/>
      </xsl:on-completion>
      <xsl:variable name="newBalance"
                    select="$balance + xs:decimal(@value)"/>
      <xsl:variable name="thisDate" select="xs:date(@date)"/>
      <xsl:if test="exists($prevDate) and $thisDate ne $prevDate">
        <balance date="${prevDate}"
                  value="{format-number($balance, '0.00')}"/>
      </xsl:if>
      <xsl:next-iteration>
        <xsl:with-param name="balance" select="$newBalance"/>
        <xsl:with-param name="prevDate" select="$thisDate"/>
      </xsl:next-iteration>
    </xsl:iterate>
  </xsl:source-document>
</account>

```

If the sequence of transactions is empty, this code outputs a single element: `<balance date="" value="0.00"/>`.

Example: Collecting Multiple Values in a Single Pass

Problem: Given a sequence of `employee` elements, find the employees having the highest and lowest salary, while processing each employee only once.

Solution:

```

<xsl:source-document streamable="yes" href="si-iterate-035.xml">
  <xsl:iterate select="employees/employee">
    <xsl:param name="highest" as="element(employee)*"/>
    <xsl:param name="lowest" as="element(employee)*"/>
    <xsl:on-completion>
      <highest-paid-employees>
        <xsl:value-of select="$highest/name"/>
      </highest-paid-employees>
      <lowest-paid-employees>
        <xsl:value-of select="$lowest/name"/>
      </lowest-paid-employees>
    </xsl:on-completion>
    <xsl:variable name="this" select="copy-of()"/>
    <xsl:variable name="is-new-highest" as="xs:boolean"
      select="empty($highest[@salary ge current()/@salary])"/>
    <xsl:variable name="is-equal-highest" as="xs:boolean"
      select="exists($highest[@salary eq current()/@salary])"/>
    <xsl:variable name="is-new-lowest" as="xs:boolean"
      select="empty($lowest[@salary le current()/@salary])"/>
    <xsl:variable name="is-equal-lowest" as="xs:boolean"
      select="exists($lowest[@salary eq current()/@salary])"/>
    <xsl:variable name="new-highest-set" as="element(employee)*"
      select="if ($is-new-highest) then $this
        else if ($is-equal-highest) then ($highest, $this)
        else $highest"/>
    <xsl:variable name="new-lowest-set" as="element(employee)*"
      select="if ($is-new-lowest) then $this
        else if ($is-equal-lowest) then ($lowest, $this)
        else $lowest"/>
    <xsl:next-iteration>
      <xsl:with-param name="highest" select="$new-highest-set"/>
      <xsl:with-param name="lowest" select="$new-lowest-set"/>
    </xsl:next-iteration>
  </xsl:iterate>
</xsl:source-document>

```

If the input sequence is empty, this code outputs an empty `highest-paid-employees` element and an empty `lowest-paid-employees` element.

Example: Processing the Last Item in a Sequence Specially

When streaming, it is not possible to determine whether the item being processed is the last in a sequence without reading ahead. The `last`^{FO30} function therefore cannot be used in `guaranteed-streamable` code. The `xsl:iterate` instruction provides a solution to this problem.

Problem: render the last paragraph in a section in some special way, for example by using bold face. (The actual rendition is achieved by processing the paragraph with mode `last-para`.)

The solution uses `xsl:iterate` together with the `copy-of` function to maintain a one-element look-ahead by explicit coding:

```
<xsl:template match="section" mode="streaming">
  <xsl:iterate select="para">
    <xsl:param name="prev" select="()" as="element(para)?"/>
    <xsl:on-completion>
      <xsl:apply-templates select="$prev" mode="last-para"/>
    </xsl:on-completion>
    <xsl:if test="$prev">
      <xsl:apply-templates select="$prev"/>
    </xsl:if>
    <xsl:next-iteration>
      <xsl:with-param name="prev" select="copy-of(.)"/>
    </xsl:next-iteration>
  </xsl:iterate>
</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.

XSLT 3.0 also supports `xsl:try` and `xsl:catch` which define conditional processing to handle `dynamic errors`.

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`^{XP30} of the expression in the `test` attribute. The rules for determining the effective boolean value of an expression are given in [\[XPath 3.0\]](#): 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 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^{XP30}](#) of the `expression` in its `test` attribute is `true`. The rules for determining the effective boolean value of an expression are given in [\[XPath 3.0\]](#); 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>
```

8.3 Try/Catch

The `xsl:try` instruction can be used to trap dynamic errors occurring within the expression it wraps; the recovery action if such errors occur is defined using a child `xsl:catch` element.

```
<!-- Category: instruction -->
<xsl:try
  select? = expression
  rollback-output? = boolean >
  <!-- Content: (sequence-constructor, xsl:catch, (xsl:catch | xsl:fallback))* -->
</xsl:try>
```

Note:

Because a sequence constructor may contain an [xsl:fallback](#) element, the effect of this content model is that an [xsl:fallback](#) instruction may appear as a child of [xsl:try](#) in any position.

```
<xsl:catch
  errors? = tokens
  select? = expression >
  <!-- Content: sequence-constructor -->
</xsl:catch>
```

An [xsl:try](#) instruction evaluates either the expression contained in its `select` attribute, or its contained [sequence constructor](#), and returns the result of that evaluation if it succeeds without error. If a [dynamic error](#) occurs during the evaluation, the processor evaluates the first [xsl:catch](#) child element applicable to the error, and returns that result instead.

If the [xsl:try](#) element has a `select` attribute, then it **MUST** have no children other than [xsl:catch](#) and [xsl:fallback](#). That is, the `select` attribute and the contained sequence constructor are mutually exclusive. If neither is present, the result of the [xsl:try](#) is an empty sequence (no dynamic error can occur in this case).

The `rollback-output` attribute is described in [8.3.1 Recovery of Result Trees](#). The default value is `yes`.

[ERR XTSE3140] It is a [static error](#) if the `select` attribute of the [xsl:try](#) element is present and the element has children other than [xsl:catch](#) and [xsl:fallback](#) elements.

Any [xsl:fallback](#) children of the [xsl:try](#) element are ignored by an XSLT 3.0 processor, but can be used to define the recovery action taken by an XSLT 1.0 or XSLT 2.0 processor operating with [forwards compatible behavior](#).

The [xsl:catch](#) element has an optional `errors` attribute, which lists the error conditions that the [xsl:catch](#) element is designed to intercept. The default value is `errors="*"`, which catches all errors. The value is a whitespace-separated list of [NameTests^{XP30}](#); an [xsl:catch](#) element catches an error condition if this list includes a [NameTest](#) that matches the error code associated with that error condition.

Note:

Error codes are QNames. Those defined in this specification and in related specifications are all in the [standard error namespace](#), and may therefore be caught using an [`xsl:catch`](#) element such as `<xsl:catch errors="err:F0DC0001 err:F0DC0005">` where the namespace prefix `err` is bound to this namespace. Errors defined by implementers, and errors raised by an explicit call of the [`error`](#)^{FO30} function or by use of the [`xsl:message`](#) or [`xsl:assert`](#) instruction, may use error codes in other namespaces.

If more than one [`xsl:catch`](#) element matches an error, the error is processed using the first one that matches, in document order. If no [`xsl:catch`](#) matches the error, then the error is not caught (that is, evaluation of the [`xsl:try`](#) element fails with the dynamic error).

An [`xsl:catch`](#) element may have either a `select` attribute, or a contained [sequence constructor](#).

[ERR XTSE3150] It is a [static error](#) if the `select` attribute of the [`xsl:catch`](#) element is present unless the element has empty content.

The result of evaluating the [`xsl:catch`](#) element is the result of evaluating the XPath expression in its `select` attribute or the result of evaluating the contained sequence constructor; if neither is present, the result is an empty sequence. This result is delivered as the result of the [`xsl:try`](#) instruction.

If a dynamic error occurs during the evaluation of [`xsl:catch`](#), it causes the containing [`xsl:try`](#) to fail with this error. The error is not caught by other sibling [`xsl:catch`](#) elements within the same [`xsl:try`](#) instruction, but it may be caught by an [`xsl:try`](#) instruction at an outer level, or by an [`xsl:try`](#) instruction nested within the [`xsl:catch`](#).

Within the `select` expression, or within the sequence constructor contained by the [`xsl:catch`](#) element, a number of variables are implicitly declared, giving information about the error that occurred. These are lexically scoped to the [`xsl:catch`](#) element. These variables are all in the [standard error namespace](#), and they are initialized as described in the following table:

Variables Available within `xsl:catch`

Variable	Type	Value
<code>err:code</code>	<code>xs:QName</code>	The error code
<code>err:description</code>	<code>xs:string?</code>	A description of the error condition; an empty sequence if no description is available (for example, if the <code>error</code> ^{FO30} function was called with one argument).
<code>err:value</code>	<code>item()*</code>	Value associated with the error. For an error raised by calling the <code>error</code> ^{FO30} function, this is the value of the third argument (if supplied). For an error raised by evaluating <code>xsl:message</code> with <code>terminate="yes"</code> , or a failing <code>xsl:assert</code> , this is the document node at the root of the tree containing the XML message body.

Variable	Type	Value
err:module	xs:string?	The URI (or system ID) of the stylesheet module containing the instruction where the error occurred; an empty sequence if the information is not available.
err:line-number	xs:integer?	The line number within the stylesheet module of the instruction where the error occurred; an empty sequence if the information is not available. The value MAY be approximate.
err:column-number	xs:integer?	The column number within the stylesheet module of the instruction where the error occurred; an empty sequence if the information is not available. The value MAY be approximate.

Variables declared within the sequence constructor of the [xsl:try](#) element (and not within an [xsl:catch](#)) are not visible within the [xsl:catch](#) element.

Note:

Within an [xsl:catch](#) it is possible to re-throw the error using the function call `error($err:code, $err:description, $err:value)`.

The following additional rules apply to the catching of errors:

1. All dynamic errors occurring during the evaluation of the [xsl:try](#) sequence constructor or [select](#) expression are caught (provided they match one of the [xsl:catch](#) elements).

Note:

- o This includes errors occurring in functions or templates invoked in the course of this evaluation, unless already caught by a nested [xsl:try](#).
- o It also includes (for example) errors caused by calling the [error](#)^{F030} function, or the [xsl:message](#) instruction with `terminate="yes"`, or the [xsl:assert](#) instruction, or the [xs:error](#) constructor function.
- o It does not include errors that occur while evaluating references to variables whose declaration and initialization is outside the [xsl:try](#).

2. The existence of an [xsl:try](#) instruction does not affect the obligation of the processor to signal certain errors as static errors, or its right to choose whether to signal some errors (such as [type errors](#)) statically or dynamically. Static errors are never caught.
3. Some fatal errors arising in the processing environment, such as running out of memory, may cause termination of the transformation despite the presence of an [xsl:try](#) instruction. This is [implementation-dependent](#).
4. If the sequence constructor or [select](#) expression of the [xsl:try](#) causes execution of [xsl:result-document](#), [xsl:message](#), or [xsl:assert](#) instructions and fails with a dynamic error that is caught, it is implementation-dependent whether these instructions have any externally visible effect. The processor is NOT

REQUIRED to roll back any changes made by these instructions. The same applies to any side effects caused by extension functions or extension instructions.

5. A serialization error that occurs during the serialization of a [secondary result](#) produced using [`xsl:result-document`](#) is treated as a dynamic error in the evaluation of the [`xsl:result-document`](#) instruction, and may be caught (for example by an [`xsl:try`](#) instruction that contains the [`xsl:result-document`](#) instruction). A serialization error that occurs while serializing the [principal result](#) is treated as occurring after the transformation has finished, and cannot be caught.
6. A validation error is treated as occurring in the instruction that requested validation. For example, if the stylesheet is producing XHTML output and requests validation of the entire result document by means of the attribute `validation="strict"` on the instruction that creates the outermost `html` element, then a validation failure can be caught only at that level. Although the validation error might be detected, for example, while writing a `p` element at a location where no `p` element is allowed, it is not treated as an error in the instruction that writes the `p` element and cannot be caught at that level.
7. A type error may be caught if the processor raises it dynamically; this does not affect the processor's right to raise the error statically if it chooses.

The following rules are provided to define which expression is considered to fail when a type error occurs, and therefore where the error can be caught. The general principle is that where the semantics of a construct C place requirements on the type of some subexpression, a type error is an error in the evaluation of C , not in the evaluation of the subexpression.

For example, consider the following construct:

```
<xsl:variable name="v" as="xs:integer">
  <xsl:sequence select="$foo"/>
</xsl:variable>
```

The expected type of the result of the sequence constructor is `xs:integer`; if the value of variable `$foo` turns out to be a string, then a type error will occur. It is not possible to catch this by writing:

```
<xsl:variable name="v" as="xs:integer">
  <xsl:try>
    <xsl:sequence select="$foo"/>
    <xsl:catch>...</xsl:catch>
  </xsl:try>
</xsl:variable>
```

This fails to catch the error because the [`xsl:sequence`](#) instruction is deemed to evaluate successfully; the failure only occurs when the result of this instruction is bound to the variable.

A similar rule applies to functions: if the body of a function computes a result which does not conform to the required type of the function result, it is not possible to catch this error within the function body itself; it can only be caught by the caller of the function. Similarly, if an expression used to compute an argument to a function returns a value of the wrong type for the function signature, this is not considered an error in this expression, but an error in evaluating the function call as a whole.

A consequence of these rules is that when a type error occurs while initializing a global variable (because the initializer returns a value of the wrong type, given the declared type of the variable), then this error cannot be caught.

Note:

Because processors are permitted to report type errors during static analysis, it is unwise to attempt to recover from type errors dynamically. The best strategy is generally to prevent their occurrence. For example, rather than writing `$p + 1` where `$p` is a parameter of unknown type, and then catching the type error that occurs if `$p` is not numeric, it is better first to test whether `$p` is numeric, perhaps by means of an expression such as `$p instance of my:numeric`, where `my:numeric` is a union type with `xs:double`, `xs:float`, and `xs:decimal` as its member types.

8. The fact that the application tries to catch errors does not prevent the processor from organizing the evaluation in such a way as to prevent errors occurring. For example `exists(//a[10 div . gt 5])` may still do an “early exit”, rather than examining every item in the sequence just to see if it triggers a divide-by-zero error.
9. Except as specified above, the optimizer must not rearrange the evaluation (at compile time or at run time) so that expressions written to be subject to the try/catch are evaluated outside its scope, or expressions written to be external to the try/catch are evaluated within its scope. This does not prevent expressions being rearranged, but any expression that is so rearranged must carry its try/catch context with it.

8.3.1 Recovery of Result Trees

The XSLT language is designed so that a processor that chooses to execute instructions in document order will always append nodes to the result tree in document order, and never needs to update a result tree in situ. As a result, it is normal practice for XSLT processors to stream the result tree directly to its final destination (for example, a serializer) without ever holding the tree in memory. This applies whether or not the processor is streamable, and whether or not source documents are streamed.

The language specification states (see [2.14 Error Handling](#)) that when a transformation terminates with a dynamic error, the state of persistent resources affected by the transformation (for example, serialized result documents) is [implementation-defined](#), so processors are not required to take any special steps to recover such resources to their pre-transformation state; at the same time, there is no guarantee that secondary result documents produced before the failure occurs will be in a usable state.

The situation becomes more complicated when dynamic errors occur while writing to a result tree, and the dynamic error is caught by an `xsl:try/xsl:catch` instruction. The semantics of these instructions requires that when an error occurring during the evaluation of `xsl:try` is caught, the result of the `xsl:try` instruction is the result of the relevant `xsl:catch`. To achieve this, any output written to the result tree during the execution of `xsl:try` until the point where the error occurs must effectively be undone. There are two basic strategies for achieving this: either the updates are not committed to persistent storage until the `xsl:try` instruction is completed, or the updates are written in such a way that they can be rolled back in the event of a failure.

Both these strategies are potentially expensive, and both have an adverse effect on streaming, in that they affect the amount of memory needed to transform large amounts of data. XSLT 3.0 therefore provides an option to relax the requirement to recover result trees when failures occur in the course of evaluating an `xsl:try` instruction. This option is invoked by specifying `rollback-output="no"` on the `xsl:try` instruction.

The default value of the attribute is `rollback-output="yes"`.

The effect of specifying `rollback-output="no"` on `xsl:try` is as follows: if a dynamic error occurs in the course of evaluating the `xsl:try` instruction, and if the failing construct is evaluated in [final output state](#) while writing to some result document, then it is [implementation-dependent](#) whether an attempt to catch this error using

[xsl:catch](#) will be successful. If the attempt is successful, then the [xsl:try](#) instruction succeeds, delivering the result of evaluating the [xsl:catch](#) clause, and the transformation proceeds as normal. If the attempt is unsuccessful (typically, because non-recoverable updates have already been made to the result tree), then the [xsl:try](#) instruction as a whole fails with a dynamic error. The state of this result document will then be undefined, but the transformation can ignore the failure and continue to produce other result documents, for example by wrapping the [xsl:result-document](#) instruction in an [xsl:try](#) instruction that catches the relevant error.

[ERR XTDE3530] It is a [dynamic error](#) if an [xsl:try](#) instruction is unable to recover the state of a final result tree because recovery has been disabled by use of the attribute `rollback-output="no"`.

For example, consider the following:

```
<xsl:result-document href="out.xml">
  <xsl:try rollback-output="no">
    <xsl:source-document streamable="yes" href="in.xml">
      <xsl:copy-of select=". />
    </xsl:source-document>
    <xsl:catch errors="*"/>
      <error code="${err:code}" message="${err:description}" file="in.xml"/>
    </xsl:catch>
  </xsl:try>
</xsl:result-document>
```

The most likely failure to occur here is a failure to read the streamed input file `in.xml`. In the common case where this failure is detected immediately, for example if the file does not exist or the network connection is down, no output will have been written to the result document, and the attempt to catch the error is likely to be successful. If however a failure is detected after several megabytes of data have been copied to `out.xml`, for example an XML well-formedness error in the input file, or a network failure that occurs while reading the file, recovery of the output file may be impossible. In this situation the [xsl:result-document](#) instruction will fail with a dynamic error. It is possible to catch this error, but the state of the file `out.xml` will be unpredictable.

Note that adding an [xsl:try](#) instruction as a child of [xsl:source-document](#) does not help. Any error reading the input file (such as a well-formedness error) is an error in the [xsl:source-document](#) instruction and can only be caught at that level.

When `rollback-output="no"` is specified, it is still possible to ensure recovery of errors happens predictably by evaluating the potentially-failing code in [temporary output state](#): typically, within an [xsl:variable](#). In effect the variable acts as an explicit buffer for temporary results, which is only copied to the final output if evaluation succeeds.

Note:

An application might wish to ensure that when a fatal error occurs while reading an input stream, data written to persistent storage up to the point of failure is available after the transformation terminates. Setting `rollback-output="no"` does not guarantee this, but a processor might choose to interpret this as the intent.

Changing the attribute to `rollback-output="yes"` makes the stylesheet more robust and able to handle error conditions predictably, but the cost may be substantial; for example it may be necessary to buffer the whole of the result document in memory.

8.3.2 Try/Catch Examples

Example: Catching a Divide-by-Zero Error

The following example divides an employee's salary by the number of years they have served, catching the divide-by-zero error if the latter is zero.

```
<xsl:try select="salary div length-of-service">
  <xsl:catch errors="err:FOAR0001" select="()"/>
</xsl:try>
```

Example: Catching an Error during Result-tree Validation

The following example generates a result tree and performs schema validation, outputting a warning message and serializing the invalid tree if validation fails.

```
<xsl:result-document href="out.xml">
  <xsl:variable name="result">
    <xsl:call-template name="construct-output"/>
  </xsl:variable>
  <xsl:try>
    <xsl:copy-of select="$result" validation="strict"/>
    <xsl:catch>
      <xsl:message>Warning: validation of result document failed:
        Error code: <xsl:value-of select="$err:code"/>
        Reason: <xsl:value-of select="$err:description"/>
      </xsl:message>
      <xsl:sequence select="$result"/>
    </xsl:catch>
  </xsl:try>
</xsl:result-document>
```

The reason that the result tree is constructed in a variable in this example is so that the unvalidated tree is available to be used within the `xsl:catch` element. An alternative approach would be to repeat the logic for constructing the tree:

```
<xsl:try>
  <xsl:result-document href="out.xml" validation="strict">
    <xsl:call-template name="construct-output"/>
  </xsl:result-document>
  <xsl:catch>
    <xsl:message>Warning: validation of result document failed:
      Error code: <xsl:value-of select="$err:code"/>
      Reason: <xsl:value-of select="$err:description"/>
    </xsl:message>
    <xsl:call-template name="construct-output"/>
  </xsl:catch>
</xsl:try>
```

8.4 Conditional Content Construction

The facilities described in this section are designed to make it easier to generate result trees conditionally depending on what is found in the input, without violating the rules for streamability. These facilities are available whether or not streaming is in use, but they are introduced to the language specifically to make streaming easier.

The facilities are introduced first by example:

Example: Generating a Wrapper Element for a non-Empty Sequence

The following example generates an `events` element if and only if there are one or more `event` elements. The code could be written like this:

```
<xsl:if test="exists(event)">
  <events>
    <xsl:copy-of select="event"/>
  </events>
</xsl:if>
```

However, the above code would not be [guaranteed-streamable](#), because it processes the child `event` elements more than once. To make it streamable, it can be rewritten as:

```
<xsl:where-populated>
  <events>
    <xsl:copy-of select="event"/>
  </events>
</xsl:where-populated>
```

The effect of the [`xsl:where-populated`](#) instruction, as explained later, is to avoid outputting the `events` element if it would have no children. A streaming implementation will typically hold the start tag of the `events` element in a buffer, to be sent to the output destination only if and when a child node is generated.

Example: Generating a Header and Footer only if there is Content

The following example generates an h3 element and a summary paragraph only if a list of items is non-empty. The code could be written like this:

```
<xsl:if test="exists(item-for-sale)">
  <h1>Items for Sale</h1>
</xsl:if>
<xsl:apply-templates select="item-for-sale"/>
<xsl:if test="exists(item-for-sale)">
  <p>Total value: {accumulator-before('total-value')}</p>
</xsl:if>
```

However, the above code would not be guaranteed-streamable, because it processes the child `item-for-sale` elements more than once. To make it streamable, it can be rewritten as:

```
<xsl:sequence>
  <xsl:on-non-empty>
    <h1>Items for Sale</h1>
  </xsl:on-non-empty>
  <xsl:apply-templates select="item-for-sale"/>
  <xsl:on-non-empty>
    <p>Total value: {accumulator-before('total-value')}</p>
  </xsl:on-non-empty>
</xsl:sequence>
```

The effect of the `xsl:on-non-empty` instruction, as explained later, is to output the enclosed content only if the containing sequence constructor also generates “ordinary” content, that is, if there is content generated by instructions other than `xsl:on-empty` and `xsl:on-non-empty` instructions.

Example: Generating Substitute Text when there is no Content

The following example generates a summary paragraph only if a list of items is empty. The code could be written like this:

```
<xsl:apply-templates select="item-for-sale"/>
<xsl:if test="empty(item-for-sale)">
  <p>There are no items for sale.</p>
</xsl:if>
```

However, the above code would not be [guaranteed-streamable](#), because it processes the child `item-for-sale` elements more than once (the fact that the list is empty is irrelevant, because streamability is determined statically). To make the code streamable, it can be rewritten as:

```
<xsl:sequence>
  <xsl:apply-templates select="item-for-sale"/>
  <xsl:on-empty>
    <p>There are no items for sale.</p>
  </xsl:on-empty>
</xsl:sequence>
```

The effect of the [`xsl:on-empty`](#) instruction, as explained later, is to output the enclosed content only if the containing sequence constructor generates no “ordinary” content, that is, if there is no content generated by instructions other than [`xsl:on-empty`](#) and [`xsl:on-non-empty`](#) instructions.

Note:

In some cases, similar effects can be achieved by using the [`has-children`^{FO30}](#) function, which tests whether an element has child nodes without consuming the children. However, use of [`has-children`^{FO30}](#) has the drawback that the function is unselective: it cannot be used to test whether there are any children of relevance to the application. In particular, it returns true if an element contains comments or whitespace text nodes that the application might consider to be insignificant.

Note:

There are no special streamability rules for the three instructions [`xsl:where-populated`](#), [`xsl:on-empty`](#), or [`xsl:on-non-empty`](#). The [general streamability rules](#) apply. In many cases the [`xsl:on-empty`](#) and [`xsl:on-non-empty`](#) instructions will generate content that does not depend on the source document, and they will therefore be [motionless](#), but this is not required.

8.4.1 The [`xsl:where-populated`](#) instruction

```
<!-- Category: instruction -->
<xsl:where-populated>
  <!-- Content: sequence-constructor -->
</xsl:where-populated>
```

The [xsl:where-populated](#) instruction encloses a [sequence constructor](#). The result of the instruction is established as follows:

1. The sequence constructor is evaluated in the usual way (taking into account any [xsl:on-empty](#) and [xsl:on-non-empty](#) instructions) to produce a result $\$R$.
2. The result of the instruction is the value of the expression $\$R[\text{not(deemed-empty}(.))]$ where the function `deemed-empty($item as item())` returns true if and only if `$item` is one of the following:
 - o A document or element node that has no children.

Note:

If an element has attributes or namespaces, these do not prevent the element being deemed empty.

If a document or element node has children, the node is not deemed empty, even if the children are empty. For example, a document node created using an [xsl:variable](#) instruction in the form `<xsl:variable name="temp"><a/></xsl:variable>` is not deemed empty, even though the contained `<a/>` element is empty.

- o A node, other than a document or element node, whose string value is zero-length.

Note:

A whitespace-only text node is not deemed empty.

- o An atomic value such that the result of casting the atomic value to a string is zero-length.

Note:

This can happen only when the atomic value is of type `xs:string`, `xs:anyURI`, `xs:untypedAtomic`, `xs:hexBinary`, or `xs:base64Binary`.

- o A map whose size (number of key/value pairs) is zero.
- o An array (see [27.7.1 Arrays](#)) where the result of flattening the array using the [array:flatten](#)^{F031} function is either an empty sequence, or a sequence in which every item is deemed empty (applying these rules recursively).

Example: Generating an HTML list

The following example generates an HTML unnumbered list, if and only if the list is non-empty. Note that the presence of the `class` attribute does not make the list non-empty. The code is written to be streamable.

```
<xsl:where-populated>
  <ul class="my-list">
    <xsl:for-each select="source-item">
      <li><xsl:value-of select=". /></li>
    </xsl:for-each>
  </ul>
</xsl:where-populated>
```

8.4.2 The `xsl:on-empty` instruction

```
<!-- Category: instruction -->
<xsl:on-empty
  select? = expression >
  <!-- Content: sequence-constructor -->
</xsl:on-empty>
```

The `xsl:on-empty` instruction has the same content model as `xsl:sequence`, and when it is evaluated, the same rules apply. In particular, the `select` attribute and the contained sequence constructor are mutually exclusive [see [ERR_XTSE3185](#)].

When an `xsl:on-empty` instruction appears in a sequence constructor, then:

1. It must be the only `xsl:on-empty` instruction in the sequence constructor, and
2. It must not be followed in the sequence constructor by any other `instruction`, other than `xsl:fallback`, or by a significant text node (that is, a text node that has not been discarded under the provisions of [4.3 Stripping Whitespace from the Stylesheet](#)), or by a `literal result element`. It may, however, be followed by non-instructions such as `xsl:catch` where appropriate.

[**DEFINITION:** An item is **vacuous** if it is one of the following: a zero-length text node; a document node with no children; an atomic value which, on casting to `xs:string`, produces a zero-length string; or (when XPath 3.1 is supported) an array which on flattening using the `array:flatten`^{FO31} function produces either an empty sequence or a sequence consisting entirely of `vacuous` items.]

An `xsl:on-empty` instruction is triggered only if every preceding sibling instruction, text node, and literal result element in the same `sequence constructor` returns either an empty sequence, or a sequence consisting entirely of `vacuous` items.

If an `xsl:on-empty` instruction is triggered, then the result of the containing `sequence constructor` is the result of the `xsl:on-empty` instruction.

Note:

This means that the (vacuous) results produced by other instructions in the sequence constructor are discarded. This is relevant mainly when the result of the sequence constructor is used for something other than constructing a node: for example if it forms the result of a function, or the value of a variable, and the function or variable specifies a required type.

When streaming, it may be necessary to buffer vacuous items in the result sequence until it is known whether the result will contain items that are non-vacuous. In many common situations, however — in particular, when the sequence constructor is being used to create the content of a node — vacuous items can be discarded immediately because they do not affect the content of the node being constructed.

Note:

In nearly all cases, the rules for [xsl:on-empty](#) are aligned with the rules for constructing complex content. If the sequence constructor within a literal result element or an [xsl:element](#) instruction includes an [xsl:on-empty](#) instruction, then the content of the element will be the value delivered by the [xsl:on-empty](#) instruction if and only if the content would otherwise be empty.

There is one minor exception to this rule: if the sequence constructor delivers multiple zero-length strings, then in the absence of the [xsl:on-empty](#) instruction the new element would contain whitespace, made up of the separators between these zero-length strings; but [xsl:on-empty](#) takes no account of these separators.

Note:

Attribute and namespace nodes created by the sequence constructor are significant; the [xsl:on-empty](#) instruction will not be triggered if such nodes are present. If this is not the desired effect, it is possible to partition the sequence constructor to change the scope of [xsl:on-empty](#), for example:

```
<ol>
  <xsl:attribute name="class" select="numbered-list"/>
  <xsl:sequence>
    <xsl:value-of select="xyz"/>
    <xsl:on-empty select="'The list is empty'"/>
  </xsl:sequence>
</ol>
```

Note:

Where the sequence constructor is a child of an instruction with an [\[xsl:\]use-attribute-sets](#) attribute, any attribute nodes created by expanding the referenced attribute set(s) are not part of the result of the sequence constructor and therefore play no role in determining whether an [xsl:on-empty](#) or [xsl:on-non-empty](#) instruction is triggered. Equally, when the sequence constructor is a child of a [literal result element](#), attribute nodes generated by expanding the attributes of the literal result element are not taken into account.

Note:

If [xsl:on-empty](#) is the only instruction in a sequence constructor then it is always evaluated.

If [xsl:on-empty](#) and [xsl:on-non-empty](#) appear in the same sequence constructor, then the rules ensure that only one of them will be evaluated.

8.4.3 The [xsl:on-non-empty](#) instruction

```
<!-- Category: instruction -->
<xsl:on-non-empty
  select? = expression >
  <!-- Content: sequence-constructor -->
</xsl:on-non-empty>
```

The [xsl:on-non-empty](#) instruction has the same content model as [xsl:sequence](#), and when it is evaluated, the same rules apply. In particular, the `select` attribute and the contained sequence constructor are mutually exclusive [see [ERR XTSE3185](#)].

An [xsl:on-non-empty](#) instruction is evaluated only if there is at least one sibling node in the same [sequence constructor](#), excluding [xsl:on-empty](#) and [xsl:on-non-empty](#) instructions, whose evaluation yields a sequence containing an item that is not [vacuous](#). If this condition applies, then all [xsl:on-non-empty](#) instructions in the containing sequence constructor are evaluated, and their results are included in the result of the containing sequence constructor in their proper positions.

Note:

The [xsl:on-non-empty](#) instruction is typically used to generate headers or footers appearing before or after a list of items, where the header or footer is to be omitted if there are no items in the list.

Note:

Unlike [xsl:on-empty](#), the [xsl:on-non-empty](#) instruction can appear anywhere in a sequence constructor, and can appear more than once.

8.4.4 Evaluating [xsl:on-empty](#) and [xsl:on-non-empty](#) Instructions

The following non-normative algorithm explains one possible strategy for streamed evaluation of a [sequence constructor](#) containing [xsl:on-empty](#) and/or [xsl:on-non-empty](#) instructions.

The algorithm makes use of the following mutable variables:

- L : a list of instructions awaiting evaluation. Initially empty.
- R : a list of items to act as the result of the evaluation. Initially empty.

- F : a boolean flag, initially false, to indicate whether any non-vacuous items have been written to R by **ordinary instructions**. The term **ordinary instruction** means any node in the sequence constructor other than an `xsl:on-empty` or `xsl:on-non-empty` instruction.

The algorithm is as follows:

1. The nodes in the sequence constructor are evaluated in document order.
2. When an `xsl:on-non-empty` instruction is encountered, then:
 - a. If F is true, the instruction is evaluated and the result is appended to R .
 - b. Otherwise, the instruction is appended to L .
3. When an **ordinary instruction** is evaluated:
 - a. The results of the evaluation are appended to R , in order.
 - b. When a non-vacuous item is about to be appended to R , and F is false, then before appending the item to R , the following actions are taken:
 - i. Any `xsl:on-non-empty` instructions in L are evaluated, in order, and their results are appended to R .
 - ii. F is set to true.
4. When an `xsl:on-empty` instruction is encountered, then:
 - a. If F is true, the instruction is ignored.
 - b. Otherwise, the existing contents of R are discarded, the instruction is evaluated, and its results are appended to R .

Note:

The need to discard items from R arises only when all the items in R are vacuous. Streaming implementations may therefore need a limited amount of buffering to retain insignificant items until it is known whether they will be needed. However, in many common cases an optimized implementation will be able to discard vacuous items such as empty text nodes immediately, because when a node is being constructed using the rules in [5.7.1 Constructing Complex Content](#) or [5.7.2 Constructing Simple Content](#), such items have no effect on the final outcome.

Otherwise, the instruction is evaluated and its results are appended to R .

5. The result of the sequence constructor is the list of items in R .

8.4.5 [A More Complex Example](#)

This example shows how the three instructions `xsl:where-populated`, `xsl:on-empty`, and `xsl:on-non-empty` may be combined.

Example: Generating a Table only if there is Content

The following example generates a table containing the names and ages of a set of students; if there are no students, it substitutes a paragraph explaining this.

```

<div id="students">
  <xsl:where-populated>
    <table>
      <xsl:on-non-empty>
        <thead>
          <tr><th>Name</th><th>Age</th></tr>
        </thead>
      </xsl:on-non-empty>
      <xsl:where-populated>
        <tbody>
          <xsl:for-each select="student/copy-of()">
            <tr>
              <td><xsl:value-of select="name"/></td>
              <td><xsl:value-of select="age"/></td>
            </tr>
          </xsl:for-each>
        </tbody>
      </xsl:where-populated>
    </table>
  </xsl:where-populated>
  <xsl:on-empty>
    <p>There are no students</p>
  </xsl:on-empty>
</div>

```

Explanation:

- The `xsl:where-populated` around the `table` element ensures that if there is no `thead` and no `tbody`, then there will be no `table`.
- The `xsl:on-non-empty` surrounding the `thead` element ensures that the `thead` element is not output unless the `tbody` element is output.
- The `xsl:where-populated` around the `tbody` element ensures that the `tbody` element is not output unless there is at least one table row (`tr`).
- The `xsl:on-empty` around the `p` element ensures that if no `table` is output, then the paragraph `There are no students` is output instead.

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, a function parameter, or an `xsl:iterate` parameter. A parameter is a variable with the additional

property that its value can be set by the caller.]

[**DEFINITION:** A variable is a binding between a name and a value. The **value** of a variable is any sequence (of nodes, atomic values, and/or function items), as defined in [\[XDM 3.0\]](#).]

[9.1 Variables](#)

```
<!-- Category: declaration -->
<!-- Category: instruction -->
<xsl:variable
  name = eqname
  select? = expression
  as? = sequence-type
  static? = boolean
  visibility? = "public" | "private" | "final" | "abstract" >
  <!-- 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 an [EQName](#), which is expanded as described in [5.1.1 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](#).

[**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.

For the effect of the `static` attribute, see [9.6 Static Variables and Parameters](#).

The `visibility` attribute **MUST NOT** be specified for a [local variable](#): that is, it is allowed only when the parent element is [xsl:stylesheet](#), [xsl:transform](#), [xsl:package](#) or [xsl:override](#).

If the `visibility` attribute is present with the value `abstract` then the `select` attribute **MUST** be absent and the contained [sequence constructor](#) **MUST** be empty. In this situation there is no [supplied value](#), and therefore the constraint that the supplied value is consistent with the required type does not apply.

[9.2 Parameters](#)

```
<!-- Category: declaration -->
<xsl:param
  name = eqname
  select? = expression
  as? = sequence-type
  required? = boolean
  tunnel? = boolean
  static? = boolean >
  <!-- Content: sequence-constructor -->
</xsl:param>
```

The [xsl:param](#) element may be used:

- As a child of [xsl:stylesheet](#) or [xsl:package](#), to define a parameter to the transformation. [Stylesheet parameters](#) are set by the calling application: see [2.3.2 Priming a Stylesheet](#).
- 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](#). [Template parameters](#) are set by means of an [xsl:with-param](#) child element of the invoking instruction, as described in [9.10 Setting Parameter Values](#).
- 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](#). [Function parameters](#) are set positionally by means of the argument list in an XPath function call.
- As a child of [xsl:iterate](#) to define a parameter that can vary from one iteration to the next. Iteration parameters always take their default values for the first iteration, and in subsequent iterations are set using an [xsl:with-param](#) child of the [xsl:next-iteration](#) instruction.

The attributes applicable to [xsl:param](#) depend on its parent element in the stylesheet, as defined by the following table:

Attributes of the xsl:param Element

Parent Element	name	select	as	required	tunnel	static
xsl:package	mandatory	optional	optional	yes no	no	yes no
xsl:stylesheet	mandatory	optional	optional	yes no	no	yes no
xsl:template	mandatory	optional	optional	yes no	yes no	no
xsl:function	mandatory	disallowed	optional	yes	no	no
xsl:iterate	mandatory	mandatory	optional	no	no	no

In the table, the entries for the `name`, `select`, and `as` attributes indicate whether the attribute must appear, is optional, or must be absent; the entries for the `required`, `tunnel`, and `static` attributes indicate the values that are permitted if the attribute is present, with the default value shown in bold. (The value `yes` can also be written `true` or `1`, while `no` can also be written `false` or `0`.)

The `name` attribute is mandatory: it specifies the name of the parameter. The value of the `name` attribute is an [EQName](#), which is expanded as described in [5.1.1 Qualified Names](#).

[ERR XTSE0580] It is a [static error](#) if the values of the `name` attribute of two sibling `xsl:param` elements represent the same [expanded QName](#).

If the `xsl:param` element has a `select` attribute, then the sequence constructor **MUST** be empty.

The `static` attribute can take the value `yes` only on [stylesheet parameters](#), and is explained in [9.5 Global Variables and Parameters](#).

Note:

Local variables may [shadow](#) template parameters and function parameters: see [9.9 Scope of Variables](#).

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 [10.1.3 Tunnel Parameters](#).

[9.2.1 The Required Type of a Parameter](#)

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](#). If the `as` attribute is omitted, then the required type is `item()*`.

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 default value is used as the supplied value as described in [9.2.2 Default Values of Parameters](#).

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.

[9.2.2 Default Values of Parameters](#)

The optional `required` attribute of `xsl:param` may be used to indicate that a [stylesheet parameter](#) or [template parameter](#) is mandatory. The only value permitted for a [function parameter](#) is `yes` (these are always mandatory), and the only value permitted for a parameter to [xsl:iterate](#) is `no` (these are always initialized to a default value).

[**DEFINITION:** A parameter is **explicitly mandatory** if it is a [function parameter](#), or if the `required` attribute is present and has the value `yes`.] If a parameter is explicitly mandatory, then the `xsl:param` element **MUST** be empty and **MUST NOT** have a `select` attribute.

If a parameter is not [explicitly mandatory](#), then it may have a default value. The default value is obtained by evaluating the [expression](#) given in the `select` attribute or the contained [sequence constructor](#), as described in [9.3 Values of Variables and Parameters](#).

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.

[**DEFINITION:** An **explicit default** for a parameter is indicated by the presence of either a `select` attribute or a non-empty sequence constructor.]

[**DEFINITION:** If a parameter that is not **explicitly mandatory** has no **explicit default** value, then it has an **implicit default** value, which is the empty sequence if there is an `as` attribute, or a zero-length string if not.]

[**DEFINITION:** If a parameter has an **implicit default** value which cannot be converted to the **required type** (that is, if it has an `as` attribute which does not permit the empty sequence), then the parameter is **implicitly mandatory**.]

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.

Various errors can arise with regard to mandatory parameters when no value is supplied. In the rules below, **non-tunnel** means: not having a `tunnel` attribute with the value `yes`.

- [ERR XTSE3520] It is a static error if a parameter to `xsl:iterate` is **implicitly mandatory**.
- [ERR XTSE0690] It is a **static error** if a `package` contains both (a) a named template named *T* that is not overridden by another named template of higher import precedence and that has an **explicitly mandatory** non-tunnel parameter named *P*, and (b) an `xsl:call-template` instruction whose `name` attribute equals *T* and that has no non-tunnel `xsl:with-param` child element whose `name` attribute equals *P*. (All names are compared as QNames.)
- [ERR XTDE0700] It is a **dynamic error** if a template that has an **explicitly mandatory** or **implicitly mandatory** parameter is invoked without supplying a value for that parameter.

This includes the following cases:

- The template is invoked using `xsl:apply-templates`, `xsl:apply-imports`, or `xsl:next-match` and there is no `xsl:with-param` child whose `name` and `tunnel` attributes match the corresponding attributes of the mandatory parameter.
- The mandatory parameter is a tunnel parameter, and the template is invoked using `xsl:call-template`, and there is no `xsl:with-param` child whose `name` and `tunnel` attributes match the corresponding attributes of the mandatory parameter.
- The template is invoked as the entry point to the transformation, either by invoking an initial mode ([2.3.3 Apply-Templates Invocation](#)) or by invoking an initial template ([2.3.4 Call-Template Invocation](#)) and no

value is supplied for the mandatory parameter by the calling application.

9.3 Values of Variables and Parameters

A [variable-binding element](#) may specify the [supplied value](#) of a [variable](#) or the default value of a [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.

Effect of Different Attribute Combinations on `xsl:variable`

select attribute	as attribute	content	Effect
present	absent	empty	Value is obtained by evaluating the <code>select</code> attribute
present	present	empty	Value is obtained by evaluating the <code>select</code> attribute, adjusted to the type required by the <code>as</code> 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 <code>as</code> attribute permits an empty sequence

select attribute	as attribute	content	Effect
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 <code>as</code> 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 a document node containing an empty element as a child:

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

The value of the following variable is a 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 a 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 with [XSLT 1.0 behavior](#), 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. The value of the [variable](#) is this newly constructed document node. 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](#)^{DM30} in [\[XDM 3.0\]](#))

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 (see [5.7.1 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, while 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`^{FO30} functions can be used to find nodes within a temporary tree, by supplying the document node at the root of the tree as an argument to the function or by making it the context node.

Example: Two-Phase Transformation

The following stylesheet uses a temporary tree as the intermediate result of a two-phase transformation, using different [modes](#) for the two phases (see [6.6 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="3.0"
    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:package](#) or [xsl:stylesheet](#) element.

[**DEFINITION:** A [top-level variable-binding element](#) declares a **global variable** that is visible everywhere (except within its own declaration, and 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 [package](#) contains more than one non-hidden 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](#) as follows:

- If the declaration appears within the [top-level package](#) (including within an [xsl:override](#) element in the top-level package), then the focus is based on the [global context item](#) if supplied, or [absent](#) otherwise.
- If the declaration appears within a [library package](#), then the focus is [absent](#).

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 the [focus](#) is [absent](#). The values of other components of the dynamic context are the initial values as defined in [5.3.3 Initializing the Dynamic Context](#) and [5.3.4 Additional Dynamic Context Components used by XSLT](#).

The [visibility](#) of a [stylesheet parameter](#) is always (implicitly) [private](#) if the parameter is [static](#), or [public](#) if the parameter is non-static.

Note:

This rule has the effect that after combining all the packages making up a stylesheet, the non-static stylesheet parameters whose values are required necessarily have distinct names, which simplifies the design of APIs.

For the effect of the [static](#) attribute, see [9.6 Static Variables and Parameters](#).

The [visibility](#) attribute **MUST NOT** be specified for a local variable: that is, it is allowed only when the parent element is [xsl:package](#), [xsl:stylesheet](#), [xsl:transform](#), or [xsl:override](#).

If the [visibility](#) attribute is present with the value [abstract](#) then the [select](#) attribute **MUST** be absent and the contained [sequence constructor](#) **MUST** be empty. In this situation there is no [supplied value](#), and therefore the constraint that the supplied value is consistent with the required type does not apply.

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 Static Variables and Parameters

Static variables and parameters are global variables and can be used in the same way as other global variables. In addition, they can be used in [xsl:]use-when expressions and in shadow attributes.

[**DEFINITION:** A [top-level variable-binding element](#) having the attribute `static="yes"` declares a **static variable**: that is, a [global variable](#) whose value is known during static analysis of the stylesheet.]

[**DEFINITION:** A [static variable](#) declared using an [`xsl:param`](#) element is referred to as a **static parameter**.]

The `static` attribute MUST NOT take the value `yes` on an [`xsl:variable`](#) or [`xsl:param`](#) element unless it is a [top-level element](#).

When the `static` attribute is present with the value `yes`, the `visibility` attribute MUST NOT have a value other than `private`.

Note:

This rule prevents static variables being overridden in another package. Since the values of such variables may be used at compile time (for example, during processing of [xsl:]use-when expressions), the rule is necessary to ensure that packages can be independently compiled.

It is possible to make the value of a static variable or parameter available in a using package by binding a non-static public variable to its value, for example:

```
<xsl:param name="DEBUG" static="yes" select="true()"/>
<xsl:variable name="tracing" static="no" visibility="public"
select="$DEBUG"/>
```

When the attribute `static="yes"` is specified, the [`xsl:variable`](#) or [`xsl:param`](#) element MUST have empty content. In the case of [`xsl:variable`](#) the `select` attribute must be present to define the value of the variable [[see ERR_XTSE0010](#)].

If the `select` attribute is present, then it is evaluated using the rules for [static expressions](#).

The rules for the scope of static variables, and the handling of duplicate declarations, are similar to the rules for non-static variables, but with additional constraints designed to disallow forwards references. The reason for disallowing forwards references is to ensure that `use-when` attributes can always be evaluated as early as possible, and in particular to ensure that the value of a `use-when` attribute never has circular dependencies. The additional constraints are as follows:

1. The static context for evaluation of a [static expression](#) only contains those [static variables](#) visible within the containing package whose declarations occur prior to the element containing the static expression in stylesheet tree order. Stylesheet tree order is the order that results when all [`xsl:import`](#) and [`xsl:include`](#) declarations are replaced by the declarations in the imported or included stylesheet module. A static variable is not in scope within its own declaration.
2. If two static variables declared within the same package have the same name, the one that has higher [import precedence](#) is used (it is a consequence of rules defined elsewhere that there cannot be more than one declaration with highest import precedence). However, if the declaration with higher import precedence occurs after the one with lower import precedence in stylesheet tree order, then the two declarations must be consistent. For this purpose two declarations are consistent if (a) they are either both [`xsl:variable`](#)

elements, or both `xsl:param` elements, and (b) if the variables are initialized (that is, if the elements are `xsl:variable` elements, or if they are `xsl:param` elements and no value for the parameter is externally supplied) then the values of both variables must be identical^{FO30}, and must not contain function items.

Note:

This rule ensures that when a static variable reference is encountered, the value of the most recently declared static variable with that name can be used, knowing that this value cannot be overridden by a subsequent declaration having higher import precedence.

[ERR XTSE3450] It is a static error if a variable declared with `static="yes"` is inconsistent with another static variable of the same name that is declared earlier in stylesheet tree order and that has lower import precedence.

9.7 Static Expressions

[DEFINITION: A **static expression** is an XPath expression whose value must be computed during static analysis of the stylesheet.]

Static expressions appear in a number of contexts, in particular:

- In `[xsl:]use-when` attributes (see [3.13.1 Conditional Element Inclusion](#));
- In the `select` attribute of static variable declarations (`xsl:variable` or `xsl:param` with `static="yes"`);
- In shadow attributes (see [3.13.2 Shadow Attributes](#)).

There are no syntactic constraints on the XPath expression that can be used as a static expression. 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 Static Expressions

Component	Value
XPath 1.0 compatibility mode	false
Statically known namespaces	determined by the in-scope namespaces for the containing element in the stylesheet
Default element/type namespace	determined by the <code>xpath-default-namespace</code> attribute if present (see 5.1.2 Unprefixed Lexical QNames in Expressions and Patterns); otherwise the null namespace
Default function namespace	The <u>standard function namespace</u>
In-scope schema types	The type definitions that would be available in the absence of any <code>xsl:import-schema</code> declaration
In-scope element declarations	None

Component	Value
In-scope attribute declarations	None
In-scope variables	<p>The static variables visible within the containing package whose declarations occur prior to the element containing the static expression in stylesheet tree order. Stylesheet tree order is the order that results when all <code>xsl:import</code> and <code>xsl:include</code> declarations are replaced by the declarations in the imported or included stylesheet module. A static variable is not in scope within its own declaration, and it is in scope only within its declaring package, not in any using packages. If two static variables satisfying this rule have the same name and are both in scope, the one that appears most recently in stylesheet tree order is used; as a consequence of rules defined elsewhere this will always be consistent with the declaration having highest import precedence.</p>
Context item static type	Absent
Statically known function signatures	<p>The functions defined in [Functions and Operators 3.0] in the <code>fn</code> and <code>math</code> namespaces, together with:</p> <ol style="list-style-type: none"> <li data-bbox="616 945 1348 1057">1. the functions <code>element-available</code>, <code>function-available</code>, <code>type-available</code>, <code>available-system-properties</code>, and <code>system-property</code> defined in this specification; <li data-bbox="616 1080 1416 1192">2. functions that appear in both this specification and in [Functions and Operators 3.1] (for example, the functions in the <code>map</code> namespaces, and a few others such as <code>collation-key</code> and <code>json-to-xml</code>); <li data-bbox="616 1215 1337 1282">3. if XPath 3.1 is supported, functions defined in [Functions and Operators 3.1] in the <code>fn</code>, <code>math</code>, <code>map</code>, and <code>array</code> namespaces; <li data-bbox="616 1304 1082 1327">4. constructor functions for built-in types; <li data-bbox="616 1349 1411 1484">5. the set of extension functions that are present in the static context of every XPath expression (other than a static expression) within the content of the element that contains the static expression. <p>Note that stylesheet functions are <i>not</i> included in the context, which means that the function <code>function-available</code> will return <code>false</code> in respect of such functions, and <code>function-lookup</code>^{FO30} will fail to find them. The effect of this rule is to ensure that <code>function-available</code> returns true in respect of functions that can be called within the static expression. It also has the effect that these extension functions will be recognized within the static expression itself; however, the fact that a function is available in this sense gives no guarantee that a call on the function will succeed.</p>
Statically known collations	Implementation-defined
Default collation	The Unicode Codepoint Collation

<i>Component</i>	<i>Value</i>
Static Base URI	The base URI of the containing element in the stylesheet document (see Section 5.2 base-uri Accessor ^{DM30})
Statically known documents	Implementation-defined
Statically known collections	Implementation-defined
Statically known default collection type	Implementation-defined
Statically known decimal formats	A single unnamed decimal format equivalent to the decimal format that is created by an xsl:decimal-format declaration with no attributes.

Dynamic Context Components for Static Expressions

<i>Component</i>	<i>Value</i>
Context item, position, and size	Absent
Variable values	A value for every variable present in the in-scope variables. For static parameters where an external value is supplied: the externally-supplied value of the parameter. In all other cases: the value of the variable as defined in 9.3 Values of Variables and Parameters .
Named functions	The function implementation corresponding to each function signature in the statically known function signatures
Current dateTime	Implementation-defined
Implicit timezone	Implementation-defined
Default language	Implementation-defined
Default calendar	Implementation-defined
Default place	Implementation-defined
Available documents	Implementation-defined
Available collections	Implementation-defined
Default collection	Implementation-defined
Environment variables	Implementation-defined

Within a [stylesheet module](#), all static expressions are evaluated in a single [execution scope](#)^{FO30}. This need not be the same execution scope as that used for static 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](#)^{FO30} 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.

If a [static error](#) is present in a [static expression](#), it is treated in the same way as any other static error in the stylesheet module. If a [dynamic error](#) occurs during evaluation of a static expression, it is treated as a static error in the analysis of the stylesheet, while retaining its original error code.

9.8 Local Variables and Parameters

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

An [xsl:param](#) element may also be used to create a variable binding with local scope:

- [**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](#).]
- An [xsl:param](#) element may appear as a child of an [xsl:iterate](#) instruction, before any non-[xsl:param](#) children of that element. This defines a parameter whose value may be initialized on entry to the iteration, and which may be varied each time round the iteration by use of an [xsl:with-param](#) element in the [xsl:next-iteration](#) instruction.

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.

9.9 Scope of Variables

For any [variable-binding element](#), there is a region (more specifically, a set of 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 containing [package](#) (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. (For rules regarding the visibility of the variable in other packages, see [3.5.3.1 Visibility of Components](#).)

A local [variable binding element](#) is visible for all following siblings and their descendants, with the following exceptions:

1. It is not visible in any region where it is shadowed by another variable binding.
2. It is not visible within the subtree rooted at an xsl:fallback instruction that is a sibling of the variable binding element.
3. It is not visible within the subtree rooted at an xsl:catch instruction that is a sibling of the variable binding element.

The binding is not visible for the xsl:variable or xsl:param element itself.

If a binding is visible for an element then it is visible for every attribute of that element and for every text node child of that element.

[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 [\[XPath 3.0\]](#).

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.

9.10 Setting Parameter Values

```
<xsl:with-param
  name = eqname
  select? = expression
  as? = sequence-type
  tunnel? = boolean >
  <!-- 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 an [EQName](#), which is expanded as described in [5.1.1 Qualified Names](#).

The [xsl:with-param](#) element is also used when passing parameters to an iteration of the [xsl:iterate](#) instruction, or to a dynamic invocation of an XPath expression using [xsl:evaluate](#). In consequence, [xsl:with-param](#) may appear within [xsl:apply-templates](#), [xsl:apply-imports](#), [xsl:call-template](#), [xsl:evaluate](#), [xsl:next-iteration](#), and [xsl:next-match](#). (Arguments to [stylesheet functions](#), however, are supplied as part of an XPath function call: see [10.3 Stylesheet Functions](#).)

[ERR XTSE0670] It is a [static error](#) if two or more sibling [xsl:with-param](#) elements have name attributes that represent the same [expanded QName](#).

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.

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 its parent [instruction](#).

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 [10.1.3 Tunnel Parameters](#). They are used only when passing parameters to templates: for an [`xsl:with-param`](#) element that is a child of [`xsl:evaluate`](#) or [`xsl:next-iteration`](#) the `tunnel` attribute MUST either be omitted or take the value no.

In other cases it is a [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. [see [ERR_XTDE0700](#)]

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

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

Example: Circularity involving Attribute Sets

An attribute set is circular if its `use-attribute-sets` attribute references itself, directly or indirectly. So the following definitions establish a circularity:

```
<xsl:attribute-set name="a" use-attribute-sets="b"/>
<xsl:attribute-set name="b" use-attribute-sets="a"/>
```

Because attribute sets can invoke functions, global variables, or templates, and can also include instructions such as literal result elements that themselves invoke attribute sets, examples of circularity involving attribute sets can be more complex than this simple example illustrates. It is also possible to construct examples in which self-reference among attribute sets could be regarded as (terminating or non-terminating) recursion. However, because such self-references have no practical utility, any requirement to evaluate an attribute set in the course of its own evaluation is considered an error.

Note:

In previous versions of this specification, self-reference among attribute sets was defined as a static error. In XSLT 3.0 it is not always detectable statically, because attribute sets can bind to each other across package boundaries. Nevertheless, in cases where a processor can detect a static circularity, it can report this error during the analysis phase, under the general provision for reporting dynamic errors during stylesheet analysis if execution can never succeed.

[ERR XTDE0640] In general, a [circularity](#) in a [stylesheet](#) is a [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>
```

Although a circularity is detected as a dynamic error, there is no unique instruction whose evaluation triggers the error condition, and the result of any attempt to catch the error using an `xsl:try` instruction is therefore [implementation-dependent](#).

Circularities usually involve global variables or parameters, but they can also exist between [key](#) definitions (see [20.2 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.

The requirement to report a circularity as a dynamic error overrides the rule that dynamic errors in evaluating [patterns](#) are normally masked (by treating the pattern as not matching).

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](#)).

[**DEFINITION:** The following [constructs](#) are classified as **invocation constructs**: the instructions [`xsl:call-template`](#), [`xsl:apply-templates`](#), [`xsl:apply-imports`](#), and [`xsl:next-match`](#); XPath function calls that bind to [stylesheet functions](#); XPath dynamic function calls; the functions [`accumulator-before`](#) and [`accumulator-after`](#); the `[xsl:]use-attribute-sets` attribute. These all have the characteristic that they can cause evaluation of constructs that are not lexically contained within the calling construct.]

10.1 Named Templates

```
<!-- Category: instruction -->
<xsl:call-template
  name = eqname >
  <!-- 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 an [EQName](#), which is expanded as described in [5.1.1 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` and `visibility` attributes on an [`xsl:template`](#) element have no effect when the template is invoked by an [`xsl:apply-templates`](#) instruction.

[ERR XTSE0650] It is a [static error](#) if a [package](#) contains an [`xsl:call-template`](#) instruction whose `name` attribute does not match the `name` attribute of any [named template](#) visible in the containing [package](#) (this includes any template defined in this package, as well as templates accepted from used packages whose visibility in this package is not `hidden`). For more details of the process of binding the called template, see [3.5.3.5 Binding References to Components](#).

[ERR XTSE0660] It is a [static error](#) if a [package](#) contains more than one non-hidden [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 established using the binding rules described in [3.5.3.5 Binding References to Components](#). This will always be a template whose `name` attribute matches the `name` attribute of the [xsl:call-template](#) instruction. It may be a template defined in the same package that has higher [import precedence](#) than any other template with this name, or it may be a template accepted from a used package, or (if the template is not defined as `private` or `final`) it may be an overriding template in a package that uses the containing package. 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](#)).

The template name `xsl:initial-template` is specially recognized in that it provides a default entry point for stylesheet execution (see [2.3 Initiating a Transformation](#).)

10.1.1 Declaring the Context Item for a Template

The [xsl:context-item](#) element is used as a child of [xsl:template](#), to declare the required type of the context item. It is intended particularly for use when the containing template is called using an [xsl:call-template](#) instruction, but it also constrains the context item if the same template is invoked using [xsl:apply-templates](#), [xsl:apply-imports](#), or [xsl:next-match](#).

```
<xsl:context-item
    as? = item-type
    use? = "required" | "optional" | "absent" />
```

If the `as` attribute is present then its value must be an [ItemType](#)^{XP30}. If the attribute is omitted this is equivalent to specifying `as="item()"`.

[ERR XTSE3088] It is a [static error](#) if the `as` attribute is present when `use="absent"` is specified.

A [type error](#) is signaled if the supplied context item does not match its required type. No attempt is made to convert the context item to the required type (using the function conversion rules or otherwise). The error code is the same as for [xsl:param](#): [see [ERR XTTE0590](#)].

If an [xsl:context-item](#) element is present as the first child element of [xsl:template](#), it defines whether the template requires a context item to be supplied, and if so, what the type of the context item must be. If this template is the [initial named template](#), then this has the effect of placing constraints on the [global context item](#) for the transformation as a whole.

The `use` attribute of [xsl:context-item](#) takes the value `required`, `optional`, or `absent`. The default is `optional`.

If the containing [xsl:template](#) element has no `name` attribute then the only permitted value is `required`.

- If the value `required` is specified, then there must be a context item. (This will automatically be the case if the template is invoked using [xsl:apply-templates](#), [xsl:apply-imports](#), or [xsl:next-match](#), but not if it is invoked using [xsl:call-template](#).)
- If the value `optional` is specified, or if the attribute is omitted, or if the [xsl:context-item](#) element is omitted, then there may or may not be a context item when the template is invoked.
- If the value `absent` is specified, then the contained sequence constructor, and any [xsl:param](#) elements, are evaluated with an absent focus.

Note:

It is not an error to call such a template with a non-absent focus; the context item is simply treated as absent. This option is useful when streaming, since an [xsl:call-template](#) instruction may become streamable if the referenced template is declared to make no use of the context item.

The processor MAY signal a [type error](#) statically if the required context item type is incompatible with the `match` pattern, that is, if no item that satisfies the match pattern can also satisfy the required context item type.

The [xsl:context-item](#) element plays no part in deciding whether and when the template rule is invoked in response to an [xsl:apply-templates](#) instruction.

[ERR XTTE3090] It is a [type error](#) if the [xsl:context-item](#) child of [xsl:template](#) specifies that a context item is required and none is supplied by the caller, that is, if the context item is absent at the point where [xsl:call-template](#) is evaluated.

10.1.2 Passing Parameters to Named Templates

Parameters are passed to named templates using the [xsl:with-param](#) element as a child of the [xsl:call-template](#) instruction.

[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 non-tunnel [template parameter](#) named *x*, unless the [xsl:call-template](#) instruction is processed with [XSLT 1.0 behavior](#). 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 [10.1.3 Tunnel Parameters](#).

Example: Calling a Named Template with a Parameter

This example defines a named template for a `numbered-block` with a parameter 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>
```

10.1.3 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 the dynamically scoped variables found in some functional programming languages (for example, early versions of LISP), where evaluating a variable reference involves searching down the dynamic call stack for a matching variable name. There are two main use cases for the feature:

1. They provide a way to supply context information that might be needed by many templates (for example, the fact that the output is to be localized for a particular language), but which cannot be placed in a global variable because it might vary from one phase of processing to another. Passing such information using conventional parameters is error-prone, because a single [xsl:apply-templates](#) or [xsl:call-template](#) instruction that neglects to pass the information on will lead to failures that are difficult to diagnose.
2. They are particularly useful when writing a customization layer for an existing stylesheet. For example, if you want to override a template rule that displays chemical formulae, you might want the new rule to be parameterized so you can apply the house-style of a particular scientific journal. Tunnel parameters allow you to pass this information to the overriding template rule without requiring modifications to all the intermediate template rules. Again, a global variable could be used, but only if the same house-style is to be used for all chemical formulae processed during a single transformation.

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.

Tunnel parameters are passed unchanged through a built-in template rule (see [6.7 Built-in Template Rules](#)).

If a tunnel parameter is declared in an `xsl:param` element with the attribute `tunnel="yes"`, and if the parameter is explicitly or implicitly mandatory, then a dynamic error occurs [see [ERR_XTDE0700](#)] if the set of tunnel parameters passed to the template does not include a parameter with a matching expanded QName.

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 = eqname
  use-attribute-sets? = eqnames
  visibility? = "public" | "private" | "final" | "abstract"
  streamable? = boolean >
  <!-- Content: xsl:attribute* -->
</xsl:attribute-set>

```

Attribute sets generate named collections of attributes that can be used repeatedly on different constructed elements. The [xsl:attribute-set](#) declaration is used to declare attribute sets. The REQUIRED `name` attribute specifies the name of the attribute set. The value of the `name` attribute is an EQName, which is expanded as described in [5.1.1 Qualified Names](#).

[DEFINITION: An **attribute set** is defined as a set of [xsl:attribute-set](#) declarations in the same [package](#) that share the same [expanded QName](#).]

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.

10.2.1 Using Attribute Sets

[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 an [EQName](#), which is expanded as described in [5.1.1 Qualified Names](#).

[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 [EQNames](#), or if it contains an EQName that does not match the `name` attribute of any [xsl:attribute-set](#) declaration in the containing [package](#).

An [attribute set](#) may be considered as comprising a sequence of instructions, each of which is either an [xsl:attribute](#) instruction or an [attribute set invocation](#). Starting with the declarations making up an attribute set, this sequence of instructions can be generated by the following rules:

1. The relevant attribute set [declarations](#) (that is, all declarations of attribute sets within a package sharing the same [expanded QName](#)) are considered in order: first in increasing order of [import precedence](#), and within each precedence, in [declaration order](#).
2. Each declaration is expanded to a sequence of instructions as follows. First, one [attribute set invocation](#) is generated for each EQName present in the `use-attribute-sets` attribute, if present, retaining the order in which the EQNames appear. This is followed by the sequence of contained [xsl:attribute](#) instructions, in order.

[DEFINITION: An **attribute set invocation** is a pseudo-instruction corresponding to a single EQName appearing within an `[xsl:]use-attribute-sets` attribute; the effect of the pseudo-instruction is to cause the referenced [attribute set](#) to be evaluated.]

Similarly, an `[xsl:]use-attribute-sets` attribute of an `xsl:copy`, `xsl:element`, or `xsl:attribute-set` element, or of a literal result element, is expanded to a sequence of [attribute set invocations](#), one for each QName in order.

An [attribute set](#) is a named [component](#), and the binding of QNames appearing in an [attribute set invocation](#) to attribute set components follows the rules in [3.5.3.5 Binding References to Components](#).

The following two (mutually recursive) rules define how an `[xsl:]use-attribute-set` attribute is expanded:

1. An [attribute set](#) is evaluated by evaluating each of the contained [attribute set invocations](#) and `xsl:attribute` instructions in order, to deliver a sequence of attribute nodes.
2. An [attribute set invocation](#) is evaluated by evaluating the [attribute set](#) to which it is bound, as determined by the rules in [3.5.3.5 Binding References to Components](#).

For rules regarding cycles in attribute set declarations, see [9.11 Circular Definitions](#).

Note:

The effect of an [attribute set invocation](#) on the dynamic context is the same as the effect of an `xsl:call-template` instruction. In particular, it does not change the [focus](#). Although attribute sets are often defined with fixed values, or with values that depend only on global variables, it is possible to define an attribute set in such a way that the values of the constructed attributes are dependent on the context item.

Note:

In all cases the result of evaluating an [attribute set](#) is subsequently used to create the attributes of an element node, using the rules in [5.7.1 Constructing Complex Content](#). The effect of those rules is that when the result of evaluating the attribute set contains attributes with duplicate names, the last duplicate wins. The optimization rules allow a processor to avoid evaluating or validating an attribute if it is able to determine that the attribute will subsequently be discarded as a duplicate.

10.2.2 Visibility of Attribute Sets

The `visibility` attribute determines the potential visibility of the attribute set in packages other than the containing package. If the `visibility` attribute is present on any of the `xsl:attribute-set` declarations making up the definition of an [attribute set](#) (that is, all declarations within the same package sharing the same name), then it **MUST** be present, with the same value, on every `xsl:attribute-set` declaration making up the definition of that [attribute set](#).

If the `visibility` attribute is present with the value `abstract` then there must be no `xsl:attribute` children and no `use-attribute-sets` attribute.

10.2.3 Streamability of Attribute Sets

An [attribute set](#) may be designated as streamable by including the attribute `streamable="yes"` on each `xsl:attribute-set` declaration making up the attribute set. If any `xsl:attribute-set` declaration for an

attribute set has the attribute `streamable="yes"`, then every [xsl:attribute-set](#) declaration for that attribute set MUST have the attribute `streamable="yes"`.

An [attribute set](#) is [guaranteed-streamable](#) if all the following conditions are satisfied:

1. Every [xsl:attribute-set](#) declaration for the attribute set has the attribute `streamable="yes"`.
2. Every [xsl:attribute-set](#) declaration for the attribute set is [grounded](#) and [motionless](#) according to the analysis in [19.8.6 Classifying Attribute Sets](#).

Specifying `streamable="yes"` on an [xsl:attribute-set](#) element declares an intent that the attribute set should be streamable, either because it is [guaranteed-streamable](#), or because it takes advantage of streamability extensions offered by a particular processor. The consequences of declaring the attribute set to be streamable when it is not in fact guaranteed streamable depend on the conformance level of the processor, and are explained in [19.10 Streamability Guarantees](#).

[ERR XTSE0730] If an [xsl:attribute](#) set element specifies `streamable="yes"` then every attribute set referenced in its `use-attribute-sets` attribute (if present) must also specify `streamable="yes"`.

Note:

It is common for attribute sets to create attributes with constant values, and such attribute sets will always be grounded and motionless and therefore streamable. Although such cases are fairly simple for a processor to detect, references to attribute sets are not guaranteed streamable unless the attribute set is declared with the attribute `streamable="yes"`, which should therefore be used if interoperable streaming is required.

10.2.4 Evaluating Attribute Sets

Attribute sets are evaluated as follows:

- The [xsl:copy](#) and [xsl:element](#) instructions have a `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.

The [xsl:attribute](#) instructions are evaluated using the same [focus](#) as is used for evaluating the [sequence constructor](#) contained by 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.

Note:

The above rule means that for an [xsl:copy](#) element with a **select** attribute, the focus for evaluating any referenced attribute sets is the node selected by the **select** attribute, rather than the context item of the [xsl:copy](#) instruction.

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](#).

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.

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.

10.2.5 [Attribute Sets: Examples](#)

Example: A Simple Attribute Set

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](#) (subject to visibility rules).]

```

<!-- Category: declaration -->
<xsl:function
  name = eqname
  as? = sequence-type
  visibility? = "public" | "private" | "final" | "abstract"
  streamability? = "unclassified" | "absorbing" | "inspection" | "filter" |
  "shallow-descent" | "deep-descent" | "ascent" | eqname
  override-extension-function? = boolean
  [override]? = boolean
  new-each-time? = "yes" | "true" | "1" | "no" | "false" | "0" | "maybe"
  cache? = boolean >
  <!-- 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 an [EQName](#), which is expanded as described in [5.1.1 Qualified Names](#).

An [xsl:function](#) declaration can only appear as a [top-level](#) element in a stylesheet module.

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.

10.3.1 Function Name and Arity

The name of the function is given by the name attribute; the arguments are defined by child [xsl:param](#) elements; and the return type is defined by the as attribute. Together these definitions constitute the *function signature*.

[ERR XTSE0740] It is a [static error](#) if a [stylesheet function](#) has a name that is in no namespace.

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 name of the function must not be in a [reserved namespace](#): [see [ERR XTSE0080](#)]

[**DEFINITION:** The **arity** of a stylesheet function is the number of [xsl:param](#) elements in the function definition.] Optional arguments are not allowed.

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.

10.3.2 Arguments

The `xsl:param` elements define the formal parameters 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.

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 XTSE0760] It is a static error if an `xsl:param` child of an `xsl:function` element has either a `select` attribute or non-empty content.

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, and the errors that can occur, are defined in [\[XPath 3.0\]](#). The rules that apply are those for the case where [XPath 1.0 compatibility mode](#) is set to `false`.

If the `as` attribute is omitted, no conversion takes place and any value is accepted.

10.3.3 Function Result

The result of the function is the result of evaluating the contained [sequence constructor](#).

Within the sequence constructor, the `focus` is initially [absent](#); this means that any attempt to reference the context item, context position, or context size is a [dynamic error](#). (See [\[ERR XPDY0002\]](#)^{XP30}.)

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.

The optional `as` attribute indicates the [required type](#) of the result of the function. The value of the `as` attribute is a [SequenceType](#).

[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.

10.3.4 Visibility and Overriding of Functions

If the `visibility` attribute is present with the value `abstract` then the [sequence constructor](#) defining the function body **MUST** be empty.

The XPath specification states that 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. In XSLT 3.0, final determination of the function to be called cannot be made until all packages have been assembled: see [3.5.3.5 Binding References to Components](#).

An `xsl:function` declaration defines a [stylesheet function](#) which forms a [component](#) in its containing [package](#), unless

- there is another [stylesheet function](#) with the same name and [arity](#), and higher [import precedence](#), or
- the [override-extension-function](#) or [override](#) attribute has the value `no` and there is already a function with the same name and [arity](#) in the in-scope functions.

The [visibility](#) of the function in other packages depends on the value of the [visibility](#) attribute and other factors, as described in [3.5 Packages](#).

The optional [override-extension-function](#) 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-extension-function](#) 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-extension-function="yes"` ensures interoperable behavior: the same code will execute with all processors. Specifying `override-extension-function="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-extension-function](#) attribute does *not* affect the rules for deciding which of several [stylesheet functions](#) with the same name and [arity](#) takes precedence.

The [override](#) attribute is a [deprecated](#) synonym of [override-extension-function](#), retained for compatibility with XSLT 2.0. If both attributes are present then they **MUST** have the same value.

[ERR XTSE0770] It is a [static error](#) for a [package](#) to contain two or more [xsl:function](#) declarations with the same [expanded QName](#), the same [arity](#), and the same [import precedence](#), unless there is another [xsl:function](#) declaration with the same [expanded QName](#) and [arity](#), and a higher import precedence.

When the [xsl:function](#) declaration appears as a child of [xsl:override](#), there **MUST** be a stylesheet function with the same [expanded QName](#) and [arity](#) in the [package](#) referenced by the containing [xsl:use-package](#) element; the [visibility](#) of that function must be `public` or `abstract`, and the overriding and overridden functions **MUST** have the same argument types and result type.

10.3.5 Streamability of Stylesheet Functions

The [streamability](#) attribute of [xsl:function](#) is used to assign the function to one of a number of [streamability categories](#). The various categories, and their effect on the streamability of function calls, are described in [19.8.5 Classifying Stylesheet Functions](#).

The streamability category of a function characterizes the way in which the function processes any streamed nodes supplied in the first argument to the function. (In general, streamed nodes cannot be supplied in other arguments, unless they are atomized by the [function conversion rules](#).) The [streamability](#) attribute is therefore not applicable unless the function takes at least one argument.

[ERR XTSE3155] It is a static error if an `xsl:function` element with no `xsl:param` children has a `streamability` attribute with any value other than `unclassified`.

10.3.6 Dynamic Access to Functions

If a `stylesheet function` with a particular `expanded QName` and `arity` exists in the stylesheet, then a call to the `function-lookupFO30` function supplying that name and arity will return the function as a value. This applies only if the static context for the call on `function-lookupFO30` includes the `stylesheet function`, which implies that the function is visible in the containing package.

The `function-available` function, when called with a particular `expanded QName` and `arity`, returns true if and only if a call on `function-lookupFO30` with the same arguments, in the same static context, would return a function item.

Note:

For legacy reasons there is also a single-argument version of `function-available`, which returns true if there is a function with the given name regardless of arity.

The standard rules for `function-lookupFO30` require that if the supplied name and arity identify a context-dependent function such as `name#0FO30` or `lang#1FO30` (call it F), then the returned function value includes in its closure a copy of the static and dynamic context of the call to `function-lookupFO30`, and the context item for a subsequent dynamic call of F is taken from this saved context. In the case where the context item is a node in a streamed input document, saving the node is not possible. In this case, therefore, the context is saved with an absent focus, so the call on F will fail with a dynamic error saying that there is no context item available.

10.3.7 Determinism of Functions

Stylesheet functions have been designed to be largely deterministic: unless a stylesheet function calls some `extension function` which is itself nondeterministic, the function will return results that depend only on the supplied arguments. This property (coupled with the fact that the effect of calling extension functions is entirely `implementation-dependent`) enables a processor to implement various optimizations, such as removing invariant function calls from the body of a loop, or combining common subexpressions.

One exception to the intrinsic determinism of stylesheet functions arises because constructed nodes have distinct identity. This means that when a function that creates a new node is called, two calls on the function will return nodes that can be distinguished: for example, with such a function, `f:make-node()` is `f:make-node()` will return false.

Three classes of functions can be identified:

1. `DeterministicFO31` functions: as the term is defined in [Functions and Operators 3.1], these offer a guarantee that when a function is called repeatedly with the same arguments, it returns the same results. A classic example is the `docFO30` function, which offers the guarantee that `doc($X)` is `doc($X)`: that is, two calls supplying the same URI return the same node.

2. Proactive functions: these offer the guarantee that each invocation of the function causes a single execution of the function body, or behaves exactly as if it did so. In particular this means that when the function creates new nodes, it creates new nodes on each invocation. By default, [stylesheet functions](#) are proactive.
3. Elidable functions: these offer no guarantee of determinism, and no guarantee of proactive evaluation. If the function creates new nodes, then two calls on the function with the same arguments may or may not return the same nodes, at the implementation's discretion. Examples of elidable functions include the [\[Functions and Operators 3.1\]](#) functions [analyze-string](#)^{FO30} and [json-to-xml](#).

The `xsl:function` attribute of `xsl:function` allows a stylesheet function to be assigned to one of these three categories. The value `new-each-time="no"` means the function is deterministic; the value `new-each-time="yes"` means it is proactive; and the value `new-each-time="maybe"` means it is elidable.

The definition of [determinism](#)^{FO31} requires a definition of what it means for a function to be called twice with “the same” arguments and to return “the same” result. This is defined in [\[Functions and Operators 3.1\]](#), specifically by the definition of the term [identical](#)^{FO31}.

Processors have considerable freedom to optimize execution of stylesheets, and of function calls in particular, but the strategies that are adopted must respect the specification as to whether functions are deterministic, proactive, or elidable. For example, consider a function call that appears within an `xsl:for-each` instruction, where the supplied arguments to the function do not depend on the context item or on any variables declared within the `xsl:for-each` instruction. A possible optimization is to execute the function call only once, rather than executing it repeatedly each time round the loop (this is sometimes called loop-lifting). This optimization is safe when the function is deterministic or elidable, but it requires great care if the function is proactive; it is permitted only if the processor is able to determine that the results of stylesheet execution are equivalent to the results that would be obtained if the optimization had not been performed. Declaring a function call to be elidable (by writing `new-each-time="maybe"`) makes it more likely that an implementation will be able to apply this optimization, as well as other optimizations such as caching or memoization.

10.3.8 Memoization

The `cache` attribute is an optimization hint which the processor can use or ignore at its discretion; however it **SHOULD** be taken seriously, because it may make a difference to whether execution of a stylesheet is practically feasible or not.

The default value is `cache="no"`.

The value `cache="yes"` encourages the processor to retain memory of previous calls of this function during the same transformation and to reuse results from this memory whenever possible. The default value `cache="no"` encourages the processor not to retain memory of previous calls.

In all cases the results must respect the semantics. If a function is proactive (`new-each-time="yes"`) then caching of results may be infeasible, especially if the function result can include nodes; but it is not an error to request it, since some implementations may be able to provide caching, or analogous optimizations, even for proactive functions. (One possible strategy is to return a copy of the cached result, thus creating the illusion that the function has been evaluated anew.)

Note:

Memoization is essentially a trade-off between time and space; a memoized function can be expected to use more memory to deliver faster execution. Achieving an optimum balance may require configuring the size of the cache that is used; implementations MAY use additional [extension attributes](#) or other mechanisms to provide finer control of this kind.

Note:

Memoization of a function generally involves creating an associative table (for example, a hash map) that maps argument values to function results. To get this right, it is vital that the key for this table should correctly reflect what it means for two function calls to have “the same arguments”. Does it matter, for example, that one call passes the `xs:string` value "Paris", while another passes the `xs:untypedAtomic` value "Paris"? If the function is declared with `new-each-time="maybe"`, then the rules say that these cannot be treated as “the same arguments”: the definition of [identical](#)^{FO31} requires them to have exactly the same type as well as being equal. However, an implementation that is able to determine that all references to the argument within the function body only make use of its string value might be able to take advantage of this fact, and thus perform more efficient caching.

10.3.9 Examples of Stylesheet Functions

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="3.0"
  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>
```

Example: A Higher-Order Function

XPath 3.0 introduces the ability to pass function items as arguments to a function. A function that takes function items as arguments is known as a higher-order function.

The following example is a higher-order function that operates on any tree-structured data, for example an organization chart. Given as input a function that finds the direct subordinates of a node in this tree structure (for example, the direct reports of a manager, or the geographical subdivisions of an administrative area), it determines whether one object is present in the subtree rooted at another object (for example, whether one person is among the staff managed directly or indirectly by a manager, or whether one parcel of land is contained directly or indirectly within another parcel). The function does not check for cycles in the data.

```
<xsl:function name="f:is-subordinate" as="xs:boolean">
  <xsl:param name="superior"
             as="node()"/>
  <xsl:param name="subordinate"
             as="node()"/>
  <xsl:param name="get-direct-children"
             as="function(node()) as node()*"/>
  <xsl:sequence select="
    some $sub in $get-direct-children($superior) satisfies
    ($sub is $subordinate or
     f:is-subordinate($sub, $subordinate,
                      $get-direct-children))"/>
</xsl:function>
```

Given source data representing an organization chart in the form of elements such as:

```
<employee id="P57832" manager="P68951"/>
```

the following function can be defined to get the direct reports of a manager:

```
<xsl:function name="f:direct-reports"
              as="element(employee)*">
  <xsl:param name="manager" as="element(employee)"/>
  <xsl:sequence select="$manager/./employee
[@manager = $manager/@id]"/>
</xsl:function>
```

It is then possible to test whether one employee \$E reports directly or indirectly to another employee \$M by means of the function call:

```
f:is-subordinate($M, $E, f:direct-reports#1)
```

10.4 Dynamic XPath Evaluation

```

<!-- Category: instruction -->
<xsl:evaluate
  xpath = expression
  as? = sequence-type
  base-uri? = { uri }
  with-params? = expression
  context-item? = expression
  namespace-context? = expression
  schema-aware? = { boolean } >
  <!-- Content: (xsl:with-param | xsl:fallback)* -->
</xsl:evaluate>

```

The [xsl:evaluate](#) instruction constructs an XPath expression in the form of a string, evaluates the expression in a specified context, and returns the result of the evaluation.

The expression given as the value of the `xpath` attribute is evaluated and the result is converted to a string using the [function conversion rules](#).

[**DEFINITION:** The string that results from evaluating the expression in the `xpath` attribute is referred to as the **target expression**.]

[ERR XTDE3160] It is a [dynamic error](#) if the [target expression](#) is not a valid [expression](#) (that is, if a static error occurs when analyzing the string according to the rules of the XPath specification).

The `as` attribute, if present, indicates the required type of the result. If the attribute is absent, the required type is `item()*`, which allows any result. The result of evaluating the [target expression](#) is converted to the required type using the [function conversion rules](#). This may cause a [type error](#) if conversion is not possible. The result after conversion is returned as the result of the [xsl:evaluate](#) instruction.

The target expression may contain variable references; the values of such variables may be supplied using an [xsl:with-param](#) child instruction if the names of the variables are known statically, or using a map supplied as the value of the expression in the `with-params` attribute if the names are only known dynamically. If the `with-params` attribute is present then it must contain an expression whose value, when evaluated, is of type `map(xs:QName, item()*)` (see [21 Maps](#) for details of maps).

[ERR XTTE3165] It is a [type error](#) if the result of evaluating the expression in the `with-params` attribute of the [xsl:evaluate](#) instruction is anything other than a single map of type `map(xs:QName, item()*)`.

10.4.1 Static context for the target expression

The [static context](#)^{XP30} for the [target expression](#) is as follows:

- XPath 1.0 compatibility mode is `false`.
- Statically known namespaces and default element/type namespace:
 - if the `namespace-context` attribute is present, then its value is an [expression](#) whose required type is a single node. The expression is evaluated, and the in-scope namespaces of the resulting node are used as the statically known namespaces for the target expression. The binding for the default namespace in the in-scope namespaces is used as the default namespace for elements and types in the target expression.

[ERR XTTE3170] It is a [type error](#) if the result of evaluating the `namespace-context` attribute of the [`xsl:evaluate`](#) instruction is anything other than a single node.

- o if the `namespace-context` attribute is absent, then the in-scope namespaces of the [`xsl:evaluate`](#) instruction (with the exception of any binding for the default namespace) are used as the statically known namespaces for the target expression, and the value of the innermost `[xsl:]xpath-default-namespace` attribute, if any, is used as the default namespace for elements and types in the target expression.

Note:

XPath 3.0 allows expanded names to be written in a context-independent way using the syntax
`Q{namespace-uri}local-name`

- Default function namespace: the [standard function namespace](#).
- In-scope schema definitions: if the `schema-aware` attribute is present and has the [effective value](#) yes, then the in-scope schema definitions from the stylesheet context (that is, the schema definitions imported using [`xsl:import-schema`](#)). Otherwise, the built-in types (see [3.14 Built-in Types](#)).
- In-scope variables: the names of the in-scope variables are the union of the names appearing in the `name` attribute of the contained [`xsl:with-param`](#) elements, and the names present as keys in the map obtained by evaluating the `with-params` attribute, if present. The corresponding type is `item()*` in the case of a name found as a key in the `with-params` map, or the type named in the `as` attribute of [`xsl:with-param`](#) child (defaulting to `item()*`) otherwise.
 If a variable name is present both the static [`xsl:with-param`](#) children and also in the dynamic `with-params` map, the value from the latter takes precedence.

Note:

Variables declared in the stylesheet in [`xsl:variable`](#) or [`xsl:param`](#) elements are *not* in-scope within the target expression.

- Function signatures:

- o All functions defined in [\[Functions and Operators 3.0\]](#) in the `fn` and `math` namespaces;
- o All functions in the `fn` and `map` namespaces whose definitions appear both in this specification and also in [\[Functions and Operators 3.1\]](#);
- o If the processor supports XPath 3.1, all functions defined in [\[Functions and Operators 3.1\]](#) in the `fn`, `math`, `map`, and `array` namespaces;
- o Constructor functions for named simple types included in the in-scope schema definitions;
- o All user-defined functions present in the containing package provided their visibility is not `hidden` or `private`;
- o An [implementation-defined](#) set of [extension functions](#).

Note that this set deliberately excludes XSLT-defined functions in the [standard function namespace](#) including for example, [`key`](#), [`current-group`](#), and [`system-property`](#). A list of these functions is in [G.2 List of XSLT-defined functions](#).

- Statically known collations: the same as the collations available at this point in the stylesheet.

- Default collation: the same as the default collation defined at this point in the stylesheet (for example, by use of the [xsl:]default-collation attribute)
- Base URI: if the base-uri attribute is present, then its [effective value](#); otherwise, the base URI of the [xsl:evaluate](#) instruction.
- Statically known documents: the empty set
- Statically known collections: the empty set
- Statically known default collection type: node()*

10.4.2 Dynamic context for the target expression

The dynamic context for evaluation of the target expression is as follows:

- The context item, position, and size depend on the result of evaluating the expression in the `context-item` attribute. If this attribute is absent, or if the result is an empty sequence, then the context item, position, and size for evaluation of the target expression are all [absent](#). If the result of evaluating the `context-item` expression is a single item, then the target expression is evaluated with a [singleton focus](#) based on this item. [ERR_XTTE3210] If the result of evaluating the `context-item` expression is a sequence containing more than one item, then a [type error](#) is signaled.
- The **variable values** consists of the values bound to parameters defined either in the contained [xsl:with-param](#) elements, which are evaluated as described in [9.3 Values of Variables and Parameters](#), or in the map that results from evaluation of the expression in the `with-params` attribute; if the same QName is bound in both, the value in the `with-params` map takes precedence.
- The XSLT-specific aspects of the dynamic context described in [5.3.4 Additional Dynamic Context Components used by XSLT](#) are all [absent](#).
- The [named functions](#)^{XP30} (representing the functions accessible using [function-available](#) or [function-lookup](#)^{FO30}) include all the functions available in the static context, and may also include an additional [implementation-defined](#) set of functions that are available dynamically but not statically.
- All other aspects of the dynamic context are the same as the dynamic context for the [xsl:evaluate](#) instruction itself, except that an implementation [may](#) restrict the availability of external resources (for example, available documents) or provide options to restrict their availability, for security reasons.

Note:

For example, a processor may disallow access using the [doc](#)^{FO30} or [collection](#)^{FO30} functions to documents in local filestore.

10.4.3 The effect of the [xsl:evaluate](#) instruction

The XPath expression is evaluated in the same [execution scope](#)^{FO30} as the calling XSLT transformation; this means that the results of [deterministic](#)^{FO30} functions such as [doc](#)^{FO30} or [current-dateTime](#)^{FO30} will be consistent between the calling stylesheet and the called XPath expression.

It is a [dynamic error](#) if evaluation of the XPath expression fails with a dynamic error. The XPath-defined error code is used unchanged.

Note:

Implementations wanting to avoid the cost of repeated compilation of the same XPath expression should cache the compiled form internally.

Stylesheet authors need to be aware of the security risks associated with the use of [xsl:evaluate](#). The instruction should not be used to execute code from an untrusted source. To avoid the risk of code injection, user-supplied data should never be inserted into the expression using string concatenation, but should always be referenced by use of parameters.

10.4.4 [xsl:evaluate](#) as an optional feature

The [xsl:evaluate](#) instruction is newly introduced in XSLT 3.0. It is part of the dynamic evaluation feature, which is an optional feature of the specification (see [27.6 Dynamic Evaluation Feature](#)). An XSLT 3.0 processor **MAY** disable the feature, or allow users to disable the feature. The processor **MAY** be able to determine during static analysis whether or not the feature is available, or it **MAY** only be able to determine this during dynamic evaluation. In the first case we refer to the feature being **statically disabled**, in the second case to it being **dynamically disabled**.

If the feature is statically disabled, then:

- A call to `element-available('xsl:evaluate')` returns false, wherever it appears;
- A call to `system-property('xsl:supports-dynamic-evaluation')` returns the string "no", wherever it appears;
- If an [xsl:evaluate](#) instruction has an [xsl:fallback](#) child, fallback processing takes place;
- No static error is raised if an [xsl:evaluate](#) instruction is present in the stylesheet (an error occurs only if it is actually evaluated).

If the feature is dynamically disabled, then:

- A call to `element-available('xsl:evaluate')` appearing in a [static expression](#) (for example, in an `[xsl:]use-when` attribute) returns true;
- A call to `element-available('xsl:evaluate')` appearing anywhere else returns false;
- A call to `system-property('xsl:supports-dynamic-evaluation')` appearing in a [static expression](#) (for example, in an `[xsl:]use-when` attribute) returns the string "yes";
- A call to `system-property('xsl:supports-dynamic-evaluation')` appearing anywhere else returns the string "no";
- If an [xsl:evaluate](#) instruction has an [xsl:fallback](#) child, fallback processing takes place;
- In the absence of an [xsl:fallback](#) child, a dynamic error is raised if an [xsl:evaluate](#) instruction is evaluated. The dynamic error may be caught using [xsl:try](#) and [xsl:catch](#).

If a processor supports the dynamic evaluation feature, it is [implementation-defined](#) how the processor allows users to disable dynamic evaluation and it is implementation-defined whether the mechanism is static or dynamic.

[ERR XTDE3175] It is a [dynamic error](#) if an [xsl:evaluate](#) instruction is evaluated when use of [xsl:evaluate](#) has been statically or dynamically disabled.

In consequence of these rules, the recommended approach for stylesheet authors to write code that works whether or not `xsl:evaluate` is enabled is to use an `xsl:fallback` child instruction. For example:

```
<xsl:variable name="isValid" as="xs:boolean">
  <xsl:evaluate xpath="$validityCondition">
    <xsl:fallback><xsl:sequence select="true()"/></xsl:fallback>
  </xsl:evaluate>
</xsl:variable>
```

Note:

There may be circumstances where it is inappropriate to allow use of `xsl:evaluate`. For example:

- There may be security risks associated with the ability to execute code from an untrusted source, which cannot be inspected during static analysis.
- There may be environments where the available computing resources are sufficient to enable pre-compiled stylesheets to be executed, but not to enable XPath expressions to be compiled into executable code.

Processors that implement `xsl:evaluate` should provide mechanisms allowing calls on `xsl:evaluate` to be disabled. Implementations may disable the feature by default, and they may disable it unconditionally.

10.4.5 Examples of `xsl:evaluate`

Example: Using a Dynamic Sort Key

A common requirement is to sort a table on the value of an expression which is selected at run-time, perhaps by supplying the expression as a string-valued parameter to the stylesheet. Suppose that such an expression is supplied to the parameter:

```
<xsl:param name="sortkey" as="xs:string" select="@name"/>
```

Then the data may be sorted as follows:

```
<xsl:sort>
  <xsl:evaluate xpath="$sortkey" as="xs:string" context-item=". "/>
</xsl:sort>
```

Note the importance in this use case of caching the compiled expression, since it is evaluated repeatedly, once for each item in the list being sorted.

Example: Getting a Function if it Exists

If the [function-lookup^{FO30}](#) function were not available in the standard library, then a very similar function could be implemented like this:

```
<xsl:function name="f:function-lookup">
  <xsl:param name="name" as="xs:QName"/>
  <xsl:param name="arity" as="xs:integer"/>
  <xsl:try>
    <xsl:evaluate xpath="'Q{'
      || namespace-uri-from-QName($name)
      || '}'
      || local-name-from-QName($name)
      || '#'
      || $arity">
      <xsl:with-param name="name" as="xs:QName" select="$name"/>
      <xsl:with-param name="arity" as="xs:integer" select="$arity"/>
    </xsl:evaluate>
    <xsl:catch errors="err:XTDE3160" select="()"/>
  </xsl:try>
</xsl:function>
```

Note:

The main difference between this function and the standard [function-lookup^{FO30}](#) function is that there are differences in the functions that are visible: for example [function-lookup^{FO30}](#) gives access to user-defined functions with private visibility, whereas [xsl:evaluate](#) does not.

The [xsl:evaluate](#) instruction uses the supplied QName and arity to construct an expression of the form `Q{namespace-uri}local#arity`, which is then evaluated to return a function item representing the requested function.

11 Creating Nodes and Sequences

This section describes instructions that directly create new nodes, or sequences of nodes, atomic values, and/or function items.

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 [24.2 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 reference) 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 an [EQName](#) 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 [25.4 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](#)^{FO30} function to the value of the attribute, and the resulting attribute node must be given the [is-id](#) property.

Note:

If the attribute name is [xml:space](#), it is *not* an error when the value is something other than [default](#) or [preserve](#). Although the XML specification states that other values are erroneous, a document containing such values is well-formed; if erroneous values are to be rejected, schema validation should be used.

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 [24.2 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](#)^{DM30}) 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

Consider the following stylesheet:

```
<xsl:stylesheet 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 they are 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

The effect of an [xsl:namespace-alias](#) declaration is local to the [package](#) in which it appears: that is, it only affects the result of [literal result elements](#) within the same package.

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 within a [package](#) 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.

No error occurs if there is more than one such [xsl:namespace-alias](#) declaration having the same [literal namespace URI](#) and the same [target namespace URI](#), even if the [result-prefix](#) differs; in this case the [result-](#)

prefix used is the one that appears last in [declaration order](#).

[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="3.0"
    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="3.0">
            <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="3.0"
    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="3.0">

  <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 }
  inherit-namespaces? = boolean
  use-attribute-sets? = eqnames
  type? = eqname
  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 result of evaluating the [xsl:element](#) instruction, in usual circumstances, is the newly constructed element node.

[11.2.1 The Content of the Constructed Element Node](#)

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](#).

[11.2.2 The Name of the 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 [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 [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:element](#) 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: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 [dynamic error](#) if the [effective value](#) of the namespace attribute is not in the lexical space of the xs:anyURI datatype or if it is the string `http://www.w3.org/2000/xmlns/`.

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 term *conflict* here means any violation of the constraints defined in [\[XDM 3.0\]](#), for example the use of the same prefix to refer to two different namespaces in the element and in one of its attributes, the use of the prefix `xml` to refer to a namespace other than the XML namespace, or any use of the prefix `xmllns`.

11.2.3 [Other Properties of the Constructed Element Node](#)

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 `xmllns=""` or, in the case of XML Namespaces 1.1, `xmllns:p=""`) appearing on the child elements when the element 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.)

The values of the `nilled`, `is-id`, and `is-idrefs` properties of the new element depend on the `type` and `validation` attributes of the [xsl:element](#) instruction, as explained in [25.4 Validation](#).

11.2.4 [The Type Annotation of the 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 [25.4 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 }
  select? = expression
  separator? = { string }
  type? = eqname
  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 the `select` attribute is present then the sequence constructor must be empty, and if the sequence constructor is non-empty then the `select` attribute must be absent. If the `select` attribute is absent and the sequence constructor is empty, then the string 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 [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 [dynamic error](#) if the [effective value](#) of the `name` attribute is the string `xmllns`.

[ERR XTDE0860] In the case of an [xsl:attribute](#) instruction with no `namespace` attribute, it is a [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 [dynamic error](#) if the [effective value](#) of the `namespace` attribute is not in the lexical space of the `xs:anyURI` datatype or if it is the string `http://www.w3.org/2000/xmlns/`.

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 may subsequently be added, changed, or removed 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. The term *conflict* here means any violation of the constraints defined in [\[XDM 3.0\]](#), for example the use of the same prefix to refer to two different namespaces in the element and in one of its attributes, the use of the prefix `xml` to refer to a namespace other than the XML namespace, or any use of the prefix `xmlns`.

If the name of a constructed attribute is `xml:id`, the processor must perform attribute value normalization by effectively applying the [normalize-space](#)^{FO30} 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](#)^{FO30} 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"
  select="'http://www.w3.org/1999/XSL/Transform' />
```

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 [25.4 Validation](#).

The process of validation also determines the values of the `is-id` and `is-idrefs` properties on the new attribute node.

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 [4.3 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 [26.1 Character Maps](#)) or [disable-output-escaping](#) (see [26.2 Disabling Output Escaping](#)).

11.4.2 Creating Text Nodes Using `xsl:text`

```
<!-- Category: instruction -->
<xsl:text
  [disable-output-escaping]? = boolean >
  <!-- Content: #PCDATA -->
</xsl:text>
```

The [`xsl:text`](#) element is evaluated to construct a new text node.

If the element or one of its ancestors has an `[xsl:]expand-text` attribute, and the nearest ancestor with such an attribute has the value `yes`, then any unescaped curly brackets in the value of the element indicate the presence of [text value templates](#), which are expanded as described in [5.6.2 Text Value Templates](#).

In the absence of such an attribute, or if the effective value is `no`, 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 [4.4.2 Stripping Whitespace from a Source Tree](#)).

For the effect of the [deprecated](#) `disable-output-escaping` attribute, see [26.2 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]? = boolean >
  <!-- 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: if the `select` attribute is present then the sequence constructor must be empty, and if the sequence constructor is non-empty then the `select` attribute must be absent. If the `select` attribute is absent and the sequence constructor is empty, then the result of the instruction is a text node whose string value is zero-length. 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.

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 the instruction is processed with [XSLT 1.0 behavior](#). 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 [26.2 Disabling Output Escaping](#)

11.5 Creating Document Nodes

```
<!-- Category: instruction -->
<xsl:document
  validation? = "strict" | "lax" | "preserve" | "strip"
  type? = eqname >
  <!-- 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](#).

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 [25.1 Creating Secondary Results](#).

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 [25.4.2 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
  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 the `select` attribute is present then the sequence constructor must be empty, and if the sequence constructor is non-empty then the `select` attribute must be absent. If the `select` attribute is absent and the sequence constructor is empty, then 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="xmlstylesheet"
  select="('href="book.css"', 'type="text/css"')"/>
```

creates the processing instruction

```
<?xmlstylesheet href="book.css" type="text/css"?>
```

Note that the `xmlstylesheet` 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 [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 [26 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: if the `select` attribute is present then the sequence constructor must be empty, and if the sequence constructor is non-empty then the `select` attribute must be absent. Since the string value of a namespace node cannot be a zero-length string, either a `select` attribute or a non-empty sequence constructor 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 [dynamic error](#) if the string value of the new namespace node is not valid in the lexical space of the datatype `xs:anyURI`, or if it is the string `http://www.w3.org/2000/xmlns/`.

[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 [dynamic error](#) if the [effective value](#) of the `name` attribute is neither a zero-length string nor an [`NCNameNames`](#), or if it is `xmlns`.

[ERR XTDE0925] It is a [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 [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 initially established by the rules of the instruction that creates the element or attribute node, and in the event of conflicts, they may be changed 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 construct 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 the `select` attribute is present then the sequence constructor must be empty, and if the sequence constructor is non-empty then the `select` attribute must be absent. If the `select` attribute is absent and the sequence constructor is empty, then the string 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 `x2D` (hyphen) that is followed by another occurrence of `x2D` (hyphen) or that ends the comment.

11.9 Copying Nodes

11.9.1 Shallow Copy

```

<!-- Category: instruction -->
<xsl:copy
  select? = expression
  copy-namespaces? = boolean
  inherit-namespaces? = boolean
  use-attribute-sets? = eqnames
  type? = eqname
  validation? = "strict" | "lax" | "preserve" | "strip" >
  <!-- Content: sequence-constructor -->
</xsl:copy>

```

The [xsl:copy](#) instruction provides a way of copying a selected item. The selected item is the item selected by evaluating the `select` attribute if present, or the [context item](#) otherwise. If the selected item is a node, evaluating the [xsl:copy](#) instruction constructs a copy of the selected 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.

[ERR XTTE0945] It is a [type error](#) to use the [xsl:copy](#) instruction with no `select` attribute when the context item is absent.

If the `select` expression returns an empty sequence, the [xsl:copy](#) instruction returns an empty sequence, and the contained [sequence constructor](#) is not evaluated.

[ERR XTTE3180] It is a [type error](#) if the result of evaluating the `select` expression is a sequence of more than one item.

When the selected item is an atomic value or function item, the [xsl:copy](#) instruction returns this value. The [sequence constructor](#) is not evaluated.

When the selected 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](#) is not evaluated.

When the selected 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 selected node. The content of the new node is formed by evaluating the [sequence constructor](#) contained in the [xsl:copy](#) instruction. If the `select` attribute is present then the sequence constructor is evaluated with the selected item as the [singleton focus](#); otherwise it is evaluated using the context of the [xsl:copy](#) instruction unchanged. 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](#).

When the selected item is a document node, the `unparsed-entities` property of the existing document node is copied to the new document node.

When the selected item is an element or attribute node, the values of the `is-id`, `is-idrefs`, and `nilled` properties of the new element or attribute depend on the values of the `validation` and `type` attributes, as defined in [25.4 Validation](#).

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 [25.4 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.

When a node is copied, its base URI is copied, except when the result of the [`xsl:copy`](#) instruction is an element node having an [`xml:base`](#) attribute, in which case the base URI of the new node is taken as the value of its [`xml:base`](#) attribute, resolved if it is relative against the base URI of the [`xsl:copy`](#) instruction.

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`](#)^{FO30} 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-accumulators? = boolean
  copy-namespaces? = boolean
  type? = eqname
  validation? = "strict" | "lax" | "preserve" | "strip" />
```

The [`xsl:copy-of`](#) instruction can be used to construct a copy of a sequence of nodes, atomic values, and/or function items 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, atomic values, and/or function items. 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 values of the `is-id`, `is-idrefs`, and `nilled` properties of the new element depend on the values of the `validation` and `type` attributes, as defined in [25.4 Validation](#).

- 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). The `unparsed-entities` property of the original document node is copied to the new document node.
- 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 or a function item, 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 [25.4 Validation](#).

Errors may occur when copying namespace-sensitive elements or attributes using `validation="preserve"`. [see [ERR_XTTE0950](#)].

If removal of namespaces is requested using `copy-namespaces="no"`, then any validation that is requested is applied to the tree that remains after the relevant namespaces have been removed. This will cause validation to fail if there is namespace-sensitive content that depends on the presence of the removed namespaces.

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.

The effect of the `copy-accumulators` attribute is described in [18.2.2 Applicability of Accumulators](#).

[11.10 Constructing Sequences](#)

```
<!-- Category: instruction -->
<xsl:sequence
  select? = expression >
  <!-- Content: sequence-constructor -->
</xsl:sequence>
```

The [xsl:sequence](#) instruction may be used within a [sequence constructor](#) to construct a sequence of nodes, atomic values, and/or function items. 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 select atomic values or function items, the effect is very similar to the [xsl:copy-of](#) instruction.

The items comprising the result sequence are evaluated either using the `select` attribute, or using the contained [sequence constructor](#). These are mutually exclusive; if the instruction has a `select` attribute, then it **MUST** have no children other than [xsl:fallback](#) instructions. If there is no `select` attribute and no contained [sequence constructor](#), the result is an empty sequence.

[ERR XTSE3185] It is a [static error](#) if the `select` attribute of [xsl:sequence](#) is present and the instruction has children other than [xsl:fallback](#).

Any contained [xsl:fallback](#) instructions are ignored by an XSLT 2.0 or 3.0 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

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 sequence containing both nodes and non-nodes (atomic values or function items).

Note:

The main use case for allowing `xsl:sequence` to contain a sequence constructor is to allow the instructions within an `xsl:fork` element to be divided into groups.

It can also be used to limit the scope of local variables or of standard attributes such as `[xsl:]default-collation`.

12 Numbering

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

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](#)^{FO30} function may be more suitable.

Furthermore, formatting of integers where there is no requirement to calculate the position of a node in the document can now be accomplished using the [format-integer](#)^{FO30} function, which borrows many concepts from the [xsl:number](#) specification.

12.1 [The start-at Attribute](#)

The [effective value](#) of the `start-at` attribute must be a string representing a whitespace-separated sequence of one or more integers, each one optionally preceded by a minus sign, separated by whitespace. More specifically, the value must be a string matching the regular expression `-?[0-9]+(\s+-?[0-9]+)*`. This sequence of integers is used to **re-base** the sequence of integers being formatted. Specifically if `$S` is the sequence of integers represented by the `start-at` attribute, and `$V` is the sequence of integers to be formatted, then the following transformation is applied to `$V`:

```
for $i in 1 to count($V) return
  if ($i le count($S))
    then $V[$i] + $S[$i] - 1
    else $V[$i] + $S[last()] - 1
```

Note:

This means that if there are N integers in the `start-at` attribute, then these are used to re-base the first N numbers, while numbers after the N th are re-based using the last (N th) integer in the `start-at` attribute. If the `start-at` attribute contains more integers than are required, the surplus is ignored.

For example, if the attribute is given as `start-at="3 0 0"`, and the number sequence to be formatted is $(1, 1, 1, 1)$, then the re-based sequence is $3, 0, 0, 0$.

12.2 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 [\[XPath 3.0\]](#), 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)))`. If the `start-at` attribute is present, this sequence is then re-based as described in [12.1 The start-at Attribute](#). The resulting sequence of integers is used as the place marker to be formatted.

If the instruction is processed with [XSLT 1.0 behavior](#), 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 [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](#)^{FO30}: 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 [12.4 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>
```

12.3 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 [12.4 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-or-self::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 $\$A[. \text{ is } \$F \text{ or } . >> \$F]$

(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 resulting sequence of numbers is referred to as the [place marker](#).

If the `start-at` attribute is present, then the [place marker](#) is re-based as described in [12.1 The start-at Attribute](#).

The sequence of numbers is then converted into a string using the [effective values](#) of the attributes specified in [12.4 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, starting from the number 100:

```

<xsl:template match="note">
  <fo:block>
    <xsl:number level="any" from="chapter" format="(1) " start-at="100"/>
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>
```

12.4 Number to String Conversion Attributes

Note:

This specification is aligned with that of the `format-integer`^{FO30} function, but there are differences; for example grouping separators are part of the primary format token in `format-integer`^{FO30}, but are indicated by separate attributes in `xsl:number`.

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 (see [\[UNICODE\]](#)). 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, see [\[UNICODE\]](#)), 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 ١ (Arabic-Indic digit one) generates the sequence ۱ then ۲ then ۳ ...
- 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 format token `①` (circled digit one, ①) has a range imposed by the Unicode character repertoire (zero to 20 in Unicode versions prior to 3.2, or zero to 50 in subsequent versions). 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; its effective value MUST either be a string in the value space of `xs : language`, or a zero-length string. If no `lang` value is specified, or if the value is a zero-length string, 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 MAY use a fallback language identified by removing successive hyphen-separated suffixes from the supplied value until a supported language code is obtained; failing this, the processor uses the language that it would use if the `lang` attribute were omitted.

The optional `ordinal` attribute is used to indicate whether cardinal or ordinal numbers are required, and to select other options relating to the grammatical context of the number to be formatted. The allowed set of values is implementation-defined. If the attribute is absent, or if its value is zero-length, or if its value is `no` or `0` or `false`, then cardinal numbers appropriate to the selected language are output. If the value is `yes` or `1` or `true`, then ordinal numbers appropriate to the target language are output. Other values are implementation-defined.

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 ...`.

Note:

In some languages, the form of numbers (especially ordinal numbers) varies depending on the grammatical context: 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 cardinal or ordinal number required, in an [implementation-defined](#) way.

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 approach recommended in the previous version of this specification was to indicate the required ending, preceded by a hyphen: for example in German, appropriate values might be `ordinal="-e"`, `ordinal="-er"`, `ordinal="-es"`, `ordinal="-en"`.

Another approach, which might usefully be adopted by an implementation based on the open-source ICU localization library [\[ICU\]](#), or any other library making use of the Unicode Common Locale Data Repository [\[Unicode CLDR\]](#), is to allow the value of the attribute to be the name of a registered numbering rule set for the language in question, conventionally prefixed with a percent sign: for example, `ordinal="%spellout-ordinal-masculine"`, or `ordinal="%spellout-cardinal-year"`. (The attribute name `ordinal` in this case is a misnomer, but serves the purpose.)

Example: Ordinal Numbering in Italian

The specification `format="1" ordinal="-o" lang="it"`, if supported, should produce the sequence:

`1º 2º 3º 4º ...`

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.

The effective value of the grouping-separator attribute MAY be any string, including a zero-length string.

The effective value of the grouping-size attribute MUST be a string in the lexical space of xs:integer. If the resulting integer is positive then it defines the number of digits between adjacent grouping separators; if it is zero or negative, then no grouping separators are inserted.

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.

Format tokens for use with xsl:number

Note that Classical 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? = { language }
  order? = { "ascending" | "descending" }
  collation? = { uri }
  stable? = { boolean }
  case-order? = { "upper-first" | "lower-first" }
  data-type? = { "text" | "number" | eqname } >
  <!-- 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 datatype, 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 datatype 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 datatype. 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 `stringFO30` or `numberFO30` function as appropriate. If the `data-type` attribute has any other [effective value](#), then this value **MUST** be an [EQName](#) denoting an [expanded QName](#) with a non-absent namespace, 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 processed with [XSLT 1.0 behavior](#). With XSLT 1.0 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 [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. If the processor is able to detect the error statically, it **MAY** optionally signal it as a [static 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 3.0 and XPath 3.0 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 largely implementation-defined.

For more information about collations, see [Section 5.3 Comparison of strings^{FO30}](#) in [\[Functions and Operators 3.0\]](#). 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.

Every implementation **MUST** recognize the collation URI <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.

Furthermore, every implementation must recognize collation URIs representing tailorings of the Unicode Collation Algorithm (UCA), as described in [13.4 The Unicode Collation Algorithm](#). Although this form of collation URI must be recognized, implementations are not required to support every possible tailoring.

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 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 situation is described as an error 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** either be a string in the value space of `xs:language`, or a zero-length string. Supplying the zero-length string has the same effect as omitting the attribute. If a language is requested that is not supported, the processor **MAY** use a fallback language identified by removing successive hyphen-

separated suffixes from the supplied value until a supported language code is obtained; failing this, the processor behaves as if the `lang` attribute were omitted.

Note:

The fallback algorithm described above is identical to the rules in RFC4647 Basic Filtering used in BCP 47, and is specified in [\[RFC4647\]](#) in greater detail.

- 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).

When `lower-first` is requested, the returned collation SHOULD have the property that when two strings differ only in the case of one or more characters, then a string in which the first differing character is lower-case should precede a string in which the corresponding character is title-case, which should in turn precede a string in which the corresponding character is upper-case. When `upper-first` is requested, the returned collation SHOULD have the property that when two strings differ only in the case of one or more characters, then a string in which the first differing character is upper-case should precede a string in which the corresponding character is title-case, which should in turn precede a string in which the corresponding character is lower-case.

So, for example, if `lang="en"`, then `A a B b` are sorted with `case-order="upper-first"` and `a A b B` are sorted with `case-order="lower-first"`.

As a further example, if `lower-first` is requested, then a sorted sequence might be “MacAndrew, macintosh, macIntosh, Macintosh, MacIntosh, macintoshes, Macintoshes, McIntosh”. If `upper-first` is requested, the same sequence would sort as “MacAndrew, MacIntosh, Macintosh, macIntosh, macintosh, MacIntoshes, macintoshes, McIntosh”.

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 [5.3.1 Initializing the Static Context](#) and [3.7.1 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](#).

13.4 The Unicode Collation Algorithm

The description of the Unicode Collation Algorithm in this section is technically identical to the description found in [\[XPath 3.1\]](#). The description here is to be used by a processor that does not implement the [XPath 3.1 Feature](#); if the processor does implement the [XPath 3.1 Feature](#), the description in [\[XPath 3.1\]](#) applies.

XSLT 3.0 defines a family of collation URIs representing tailorings of the Unicode Collation Algorithm (UCA) as defined in [\[UNICODE TR10\]](#). The parameters used for tailoring the UCA are based on the parameters defined in the Locale Data Markup Language (LDML), defined in [\[UNICODE TR35\]](#).

This family of URIs use the scheme and path <http://www.w3.org/2013/collation/UCA> followed by an optional query part. The query part, if present, consists of a question mark followed by a sequence of zero or more semicolon-separated parameters. Each parameter is a keyword-value pair, the keyword and value being separated by an equals sign.

All implementations must recognize URIs in this family. This applies to all places where collations are used, including (for example) the [xsl:sort](#), [xsl:key](#), [xsl:for-each-group](#), and [xsl:merge-key](#) elements, the [\[xsl:\]default-collation](#) attribute, and the [collation](#) argument of functions such as [contains](#)^{F030}, [max](#)^{F030}, and [collation-key](#). If the [fallback](#) parameter is present with the value `no`, then the implementation **MUST** either use a collation that conforms with the rules in the Unicode specifications for the requested tailoring, or fail with a static or dynamic error indicating that it does not provide the collation (the error code should be the same as if the collation URI were not recognized). If the [fallback](#) parameter is omitted or takes the value `yes`, and if the collation URI is well-formed according to the rules in this section, then the implementation **MUST** accept the collation URI, and **SHOULD** use the available collation that most closely reflects the user's intentions. For example, if the collation URI requested is <http://www.w3.org/2013/collation/UCA?lang=se;fallback=yes> and the implementation does not include a fully conformant version of the UCA tailored for Swedish, then it **MAY** choose to use a Swedish collation that is known to differ from the UCA definition, or one whose conformance has not been established. It might even, as a last resort, fall back to using codepoint collation.

If two query parameters use the same keyword then the last one wins. If a query parameter uses a keyword or value which is not defined in this specification then the meaning is [implementation-defined](#). If the implementation recognizes the meaning of the keyword and value then it **SHOULD** interpret it accordingly; if it does not recognize the keyword or value then if the [fallback](#) parameter is present with the value `no` it should reject the collation as unsupported, otherwise it should ignore the unrecognized parameter.

The following query parameters are defined. If any parameter is absent, the default is [implementation-defined](#) except where otherwise stated. The meaning given for each parameter is non-normative; the normative specification is found in [\[UNICODE TR35\]](#).

Options for the Unicode Collation Algorithm

Keyword	Values	Meaning
<code>fallback</code>	<code>yes</code> <code>no</code> (default <code>yes</code>)	Determines whether the processor uses a fallback collation if a conformant collation is not available.
<code>lang</code>	language code, as defined for the lang attribute of xsl:sort	The language whose collation conventions are to be used.
<code>version</code>	string	The version number of the UCA to be used.
<code>strength</code>	<code>primary</code> <code>secondary</code> <code>tertiary</code> <code>quaternary</code> <code>identical</code> , or <code>1 2 3 4 5</code> as synonyms	The collation strength as defined in UCA. Primary strength takes only the base form of the character into account (so <code>A=a=Ã=â</code>); secondary strength ignores case but considers accents and diacritics as significant (so <code>A=a</code> and <code>Ã=â</code> but <code>â!=a</code>); tertiary considers case as significant (<code>A!=a!=Ã!=â</code>); quaternary considers spaces and punctuation that would otherwise be ignored (for example <code>data-base=database</code>).

Keyword	Values	Meaning
maxVariable	space punct symbol currency (default punct)	Indicates that all characters in the specified group and earlier groups are treated as "noise" characters to be handled as defined by the <code>alternate</code> parameter. For example, <code>maxVariable=punct</code> indicates that characters classified as whitespace or punctuation get this treatment.
alternate	non-ignorable shifted blanked (default non-ignorable)	Controls the handling of characters such as spaces and hyphens; specifically, the "noise" characters in the groups selected by the <code>maxVariable</code> parameter. The value <code>non-ignorable</code> indicates that such characters are treated as distinct at the primary level (so <code>data base</code> sorts before <code>datatype</code>); <code>shifted</code> indicates that they are used to differentiate two strings only at the quaternary level, and <code>blanked</code> indicates that they are taken into account only at the <code>identical</code> level.
backwards	yes no (default no)	The value <code>backwards=yes</code> indicates that the last accent in the search term is the most significant.
normalization	yes no (default no)	Indicates whether search terms are converted to normalization form D.
caseLevel	yes no (default no)	When used with primary strength, setting <code>caseLevel=yes</code> has the effect of ignoring accents while taking account of case.
caseFirst	upper lower	Indicates whether upper-case precedes lower-case or vice versa.
numeric	yes no (default no)	When <code>numeric=yes</code> is specified, a sequence of consecutive digits is interpreted as a number, for example <code>chap2</code> sorts before <code>chap12</code> .
reorder	a comma-separated sequence of reorder codes, where a reorder code is one of <code>space</code> , <code>punct</code> , <code>symbol</code> , <code>currency</code> , <code>digit</code> , or a four-letter script code defined in [ISO 15924 Register] , the register of scripts maintained by the Unicode Consortium in its capacity as registration authority for [ISO 15924] .	Determines the relative ordering of text in different scripts; for example the value <code>digit,Grek,Latn</code> indicates that digits precede Greek letters, which precede Latin letters.

Note:

This list excludes parameters that are inconvenient to express in a URI, or that are applicable only to substring matching.

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`](#)^{FO30}: see [\[Functions and Operators 3.0\]](#).

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 [`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
  composite? = boolean
  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 item 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 a [static error](#) to specify the [collation](#) attribute or the [composite](#) attribute if neither the [group-by](#) attribute nor [group-adjacent](#) attribute is specified.

[**DEFINITION:** If either of the [group-by](#) or [group-adjacent](#) attributes is present, then for each item in the [population](#) a set of **grouping keys** is calculated, as follows: the expression contained in the [group-by](#) or [group-adjacent](#) attribute is evaluated; the result is atomized; and any [xs:untypedAtomic](#) values are cast to [xs:string](#). If [composite="yes"](#) is specified, there is a single grouping key whose value is the resulting sequence; otherwise, there is a set of grouping keys, consisting of the distinct atomic values present in the result sequence.]

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](#).

If the [group-by](#) attribute is present, and if the [composite](#) attribute is omitted or takes the value `no`, then an item in the population **MAY** have multiple grouping keys: that is, the [group-by](#) expression evaluates to a sequence, and each item in the sequence is treated as a separate grouping key. 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, and if the [composite](#) attribute is omitted or takes the value `no`, then each item in the population **MUST** have exactly one grouping key value.

[ERR XTTE1100] It is a [type error](#) if the result of evaluating the [group-adjacent](#) expression is an empty sequence or a sequence containing more than one item, unless [composite="yes"](#) is specified.

[Grouping keys](#) are compared using the rules for the [deep-equal](#)^{FO30} function. This means that values of type [xs:untypedAtomic](#) will be cast to [xs:string](#) before the comparison, and that items that are not comparable using the [eq](#) operator are considered to be not equal, that is, they are allocated to different groups. It also means that the value `Nan` is considered equal to itself. 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.

[ERR XTDE1110] It is a [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](#).

The way in which an [`xsl:for-each-group`](#) element is evaluated depends on which of the four group-defining attributes is present:

- 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. If `composite="yes"` is specified, there will be a single grouping key, which will in general be a sequence of zero or more atomic values; otherwise, there will be zero or more grouping keys, each of which will be a single atomic value. For each one of these [grouping keys](#), if there is already a group created to hold items having that grouping key value, J is appended 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 appended to zero, one, or many groups. An item will never be appended 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 population contains values of different numeric types that differ from each other by small amounts, then the `eq` operator is not transitive, because of rounding effects occurring during type promotion. The effect of this is described in [14.5 Non-Transitivity](#).

- 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 appended 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](#).

The items in the [population](#) are examined in [population order](#). If an item matches the pattern, or is the first item in the population, then a new group is created and the item becomes its first member. Otherwise, the item is appended to the same group as its preceding item within the population.

- If the `group-ending-with` attribute is present, then its value **MUST** be a [pattern](#).

The items in the [population](#) are examined in [population order](#). If an item is the first item in the population, or if the previous item in the population matches the pattern, then a new group is created and the item becomes its first member. Otherwise, the item is appended to the same group as its preceding item within the population.

In all cases the order of items within each group is predictable, and reflects the original [population order](#), in that the items are processed in population order and each item is appended at the end of zero or more groups.

Note:

As always, a different algorithm may be used if it achieves the same effect.

[**DEFINITION:** For each [group](#), the item within the group that is first in [population order](#) is known as the **initial item** of the 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 group in the [processing order](#) of the groups, and the [context size](#) is the number of groups. This has the effect that within the [sequence constructor](#), a call on `position()` takes successive values 1, 2, ... `last()`.

[14.2 Accessing Information about the Current Group Value](#)

Two pieces of information are available during the processing of each group (that is, while evaluating the sequence constructor contained in the [`xsl:for-each-group`](#) instruction, and also while evaluating the sort key of a group as expressed by the `select` attribute or sequence constructor of an [`xsl:sort`](#) child of the [`xsl:for-each-group`](#) element):

- [DEFINITION: The **current group** is the [group](#) itself, as a sequence of items].
- [DEFINITION: The **current grouping key** is a single atomic value, or in the case of a composite key, a sequence of atomic values, containing the [grouping key](#) of the items in the [current group](#).]

Information about the [current group](#) and the [current grouping key](#) is held in the dynamic context, and is available using the [`current-group`](#) and [`current-grouping-key`](#) functions respectively.

In XSLT 2.0, the [current group](#) and the [current grouping key](#) were passed unchanged through calls of [`xsl:apply-templates`](#) and [`xsl:call-template`](#), and also [`xsl:apply-imports`](#) and [`xsl:next-match`](#). This behavior is retained in XSLT 3.0 except in the case where streaming is in use: specifically, if the [`xsl:apply-templates`](#), [`xsl:call-template`](#), [`xsl:apply-imports`](#), or [`xsl:next-match`](#) instruction occurs within a [declared-streamable](#) construct (typically, within an [`xsl:source-document`](#) instruction, or within a streamable [template rule](#)), then the current group and current grouping key are set to [absent](#) in the called template. The reason for this is to allow the streamability of an [`xsl:for-each-group`](#) instruction to be assessed statically, as described in [19.8.4.19 Streamability of xsl:for-each-group](#).

[14.2.1 `fn:current-group`](#)

Summary

Returns the group currently being processed by an [`xsl:for-each-group`](#) instruction.

Signature

```
fn:current-group() as item()*
```

Properties

This function is [deterministic](#)^{FO30}, [context-dependent](#)^{FO30}, and [focus-independent](#)^{FO30}.

Rules

The evaluation context for XPath [expressions](#) includes a component called the [current group](#), which is a sequence.

The function [current-group](#) returns the sequence of items making up the current group.

The current group is bound during evaluation of the [xsl:for-each-group](#) instruction. If no [xsl:for-each-group](#) instruction is being evaluated, the current group will be [absent](#): that is, any reference to it will cause a dynamic error.

The effect of [invocation constructs](#) on the [current group](#) is as follows:

- If the [invocation construct](#) is contained within a [declared-streamable construct](#) (for example, if it is within an [xsl:source-document](#) instruction with the attribute `streamable="yes"`, or within a streamable template), then the invocation construct sets the current group to [absent](#). In this situation the scope of the current group is effectively static; it can only be referenced within the body of the [xsl:for-each-group](#) instruction to which it applies.
- If the [invocation construct](#) is a (static or dynamic) function call, then the invocation construct sets the current group to [absent](#).
- Otherwise the [invocation construct](#) leaves the current group unchanged. In this situation the scope of the current group is effectively dynamic: it can be referenced within called templates and attribute sets.

The current group is initially [absent](#) during the evaluation of global variables and stylesheet parameters, during the evaluation of the `use` attribute or contained sequence constructor of [xsl:key](#), and during the evaluation of the `initial-value` attribute of [xsl:accumulator](#) and the `select` attribute of contained sequence constructor of [xsl:accumulator-rule](#).

Error Conditions

[ERR XTSE1060] It is a [static error](#) if the [current-group](#) function is used within a [pattern](#).

[ERR XTDE1061] It is a [dynamic error](#) if the [current-group](#) function is used when the current group is [absent](#), or when it is invoked in the course of evaluating a pattern. The error `MAY` be reported statically if it can be detected statically.

Notes

Like other XSLT extensions to the dynamic evaluation context, the [current group](#) is not retained as part of the closure of a function value. This means that the expression `current-group#0` is valid and returns a function value, but any invocation of this function will fail with a dynamic error [see [ERR XTDE1061](#)].

14.2.2 [fn:current-grouping-key](#)

Summary

Returns the grouping key of the group currently being processed using the [xsl:for-each-group](#) instruction.

Signature

```
fn:current-grouping-key() as xs:anyAtomicType*
```

Properties

This function is [deterministic](#)^{FO30}, [context-dependent](#)^{FO30}, and [focus-independent](#)^{FO30}.

Rules

The evaluation context for XPath [expressions](#) includes a component called the [current grouping key](#), which is a sequence of atomic values. The current grouping key is the [grouping key](#) shared in common by all the items within the [current group](#).

The function [current-grouping-key](#) returns the [current grouping key](#).

The current grouping key is bound during evaluation of an [xsl:for-each-group](#) instruction that has a group-by or group-adjacent attribute. If no [xsl:for-each-group](#) instruction is being evaluated, the current grouping key will be [absent](#), which means that any reference to it causes a dynamic error. The current grouping key is also set to [absent](#) during the evaluation of an [xsl:for-each-group](#) instruction with a group-starting-with or group-ending-with attribute.

The effect of [invocation constructs](#) on the [current grouping key](#) is as follows:

- If the [invocation construct](#) is contained within a [declared-streamable construct](#) (for example, if it is within an [xsl:source-document](#) instruction with the attribute `streamable="yes"`, or within a streamable template), then the invocation construct sets the current grouping key to [absent](#). In this situation the scope of the current group is effectively static; it can only be referenced within the body of the [xsl:for-each-group](#) instruction to which it applies.
- If the [invocation construct](#) is a (static or dynamic) function call, then the invocation construct sets the current grouping key to [absent](#).
- Otherwise the [invocation construct](#) leaves the current grouping key unchanged. In this situation the scope of the current group is effectively dynamic: it can be referenced within called templates and attribute sets.

The current grouping key is initially [absent](#) during the evaluation of global variables and stylesheet parameters, during the evaluation of the `use` attribute or contained sequence constructor of [xsl:key](#), and during the evaluation of the `initial-value` attribute of [xsl:accumulator](#) and the `select` attribute of contained sequence constructor of [xsl:accumulator-rule](#).

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 if `composite="no"` is specified (explicitly or implicitly), or a sequence of atomic values if `composite="yes"` is specified.

At other times, the current grouping key will be [absent](#).

The [grouping keys](#) of all items in a group are not necessarily identical. For example, one might be an `xs:float` while another is a numerically equal `xs:decimal`. The [current-grouping-key](#) function returns the grouping key of the [initial item](#) in the group, after atomization and casting of `xs:untypedAtomic` values to `xs:string`.

The function takes no arguments.

Error Conditions

[ERR XTSE1070] It is a [static error](#) if the [current-grouping-key](#) function is used within a [pattern](#).

[ERR XTDE1071] It is a [dynamic error](#) if the [current-grouping-key](#) function is used when the current grouping key is [absent](#), or when it is invoked in the course of evaluating a pattern. The error **MAY** be reported statically if it can be detected statically.

Notes

Like other XSLT extensions to the dynamic evaluation context, the [current grouping key](#) is not retained as part of the closure of a function value. This means that the expression `current-grouping-key#0` is valid and returns a function value, but any invocation of this function will fail with a dynamic error [see [ERR XTDE1071](#)].

14.3 Ordering among Groups

[**DEFINITION:** There is a total 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 total 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](#) if the [sort key specification](#) is [stable](#), otherwise in an [implementation-dependent](#) order.

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 [absent](#).

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())"/>`

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 and sorted, 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="3.0" 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="current-grouping-key()"/></td>
      <td>
        <xsl:for-each select="current-group()/@name">
          <xsl:sort select=". "/>
          <xsl:if test="position() ne 1">, </xsl:if>
          <xsl:value-of select=". "/>
        </xsl:for-each>
      </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 second solution is to nest one [xsl:for-each-group](#) element directly inside another. XSLT 3.0 introduces a third option, which is to define the grouping key as composite:

```
<xsl:for-each-group select="cities/city"
  group-by="@name, @country"
  composite="yes">
  <p>
    <xsl:value-of select="current-grouping-key()[1] || ', ' ||
      current-grouping-key()[2] || ':' || avg(current-group()//@pop)"/>
  </p>
</xsl:for-each-group>
```

Note:

The string concatenation operator `||` is new in XPath 3.0.

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 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 this 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>:<br/>
  <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>
```

14.5 Non-Transitivity

If the population contains values of different numeric types that differ from each other by small amounts, then the `eq` operator is not transitive, because of rounding effects occurring during type promotion. It is thus possible to have three values A , B , and C among the grouping keys of the population such that $A \neq B$, $B \neq C$, but $A = C$.

For example, this arises when computing

```
<xsl:for-each-group group-by=". " select=
  xs:float('1.0'),
  xs:decimal('1.000000000010000000001'),
  xs:double('1.00000000001')"/>
```

because the values of type `xs:float` and `xs:double` both compare equal to the value of type `xs:decimal` but not equal to each other.

In this situation the results **MUST** be equivalent to the results obtained by the following algorithm:

- For each item J in the [population](#) in [population order](#), for each of the [grouping keys](#) K for that item in sequence, the processor identifies those existing groups G such that the grouping key of the [initial item](#) of G is equal to K .
- If there is exactly one group G , then J is added to this group, unless J is already a member of this group.
- If there is no group G , then a new group is created with J as its first item.
- If there is more than one group G (which can only happen in exceptional circumstances involving non-transitivity), then one of these groups is selected in an implementation-dependent way, and J is added to this group, unless J is already a member of this group.

The effect of these rules is that (a) every item in a non-singleton group has a grouping key that is equal to that of at least one other item in that group, (b) for any two distinct groups, there is at least one pair of items (one from each group) whose grouping keys are not equal to each other.

15 Merging

The [`xsl:merge`](#) instruction allows a sorted sequence of items to be constructed by merging several input sequences. Each input sequence **MUST** have a merge key (one or more atomic values that can be computed as a function of the items in the sequence); the input sequence **MUST** either already be sorted on the value of its merge keys, or pre-sorting on these values must be requested. The merge keys for the different input sequences **MUST** be compatible in the sense that key values from an item in one sequence are always comparable with key values from an item in a different sequence.

For example, if two log files contain details of events sorted by date and time, then the [`xsl:merge`](#) instruction can be used to combine these into a single sequence that is also sorted by date and time.

The data written to the output sequence can be computed in an arbitrary way from the data in the input sequences, provided it follows the ordering of the input sequences.

The [`xsl:merge`](#) instruction can be used to merge several sequences of items that all have the same structure (more precisely, sequences whose merge keys are computed in the same way): for example, log files created by the same application running on different machines in a server farm. Alternatively, [`xsl:merge`](#) can be used to merge

sequences that have different structure (sequences whose merge keys are computed in different ways), provided that the computed merge keys are compatible: an example might be two log files created by different applications, using different XML vocabularies, that both contain timestamped events but represent the timestamp in different ways. The `xsl:merge-source` element represents a set of input sequences that follow common rules, including the rules for computing the merge key. The `xsl:merge` operation may take any number of `xsl:merge-source` elements representing different rules for input sequences, and each `xsl:merge-source` element may describe any number (zero or more) of input sequences. The number of input sequences to the merging operation is thus fixed only at the time the `xsl:merge` instruction is evaluated, and MAY vary from one evaluation to another.

The following examples illustrate some of the possibilities. The detailed explanation of the constructs used follows later in this section.

Example: Merging All the Files in a Collection

This example takes as input a homogeneous collection of XML log files each of which contains a sorted sequence of `event` elements with a `timestamp` attribute validated as an instance of `xs:dateTime`. It merges the events from the input files into a single sorted output file.

```
<xsl:result-document href="merged-events.xml">
  <events>
    <xsl:merge>
      <xsl:merge-source for-each-source="uri-collection('log-files')"
                        select="events/event">
        <xsl:merge-key select="@timestamp"/>
      </xsl:merge-source>
      <xsl:merge-action>
        <xsl:copy-of select="current-merge-group()"/>
      </xsl:merge-action>
    </xsl:merge>
  </events>
</xsl:result-document>
```

The example assumes that there are several input files each of which has a structure similar to the following, in which the `timestamp` attribute has a typed value that is an instance of `xs:dateTime`:

```
<events>
  <event timestamp="2009-08-20T12:01:01Z">Transaction T1234 started</event>
  <event timestamp="2009-08-20T12:01:08Z">Transaction T1235 started</event>
  <event timestamp="2009-08-20T12:01:12Z">Transaction T1235 ended</event>
  <event timestamp="2009-08-20T12:01:15Z">Transaction T1234 ended</event>
</events>
```

The output file will have the same structure, and will contain copies of all the `event` elements from all of the input files, in sorted order. Note that multiple events with the same timestamp can occur either within a single file or across multiple files: the order of appearance of these events in the output file corresponds to the order of the log files within the collection (which might or might not be predictable, depending on the implementation).

Example: Merging Two Heterogeneous Files

This example takes as input two log files with different structure, producing a single merged output in which the entries have a common structure:

```
<xsl:result-document href="merged-events.xml">
  <events>
    <xsl:merge>
      <xsl:merge-source select="doc('log-file-1.xml')/events/event">
        <xsl:merge-key select="@timestamp"/>
      </xsl:merge-source>
      <xsl:merge-source select="doc('log-files-2.xml')/log/day/record">
        <xsl:merge-key select="dateTime(../@date, time)"/>
      </xsl:merge-source>
      <xsl:merge-action>
        <xsl:apply-templates select="current-merge-group()"
          mode="standardize-log-entry"/>
      </xsl:merge-action>
    </xsl:merge>
  </events>
</xsl:result-document>
```

Here the first input file has a structure similar to that shown in the previous example, while the second input has a different structure, of the form:

```
<log>
  <day date="2009-08-20">
    <record>
      <time>12:01:09-05:00</time>
      <message>Temperature 15.4C</message>
    </record>
    <record>
      <time>12:03:00-05:00</time>
      <message>Temperature 18.2C</message>
    </record>
  </day>
</log>
```

The templates in mode `standardize-log-entry` convert the log entries to a common output format, for example:

```
<xsl:template match="event" mode="standardize-log-entry"
  as="schema-element(event)">
  <xsl:copy-of select=". " validation="preserve"/>
</xsl:template>

<xsl:template match="record" mode="standardize-log-entry"
  as="schema-element(event)">
  <event timestamp="{dateTime(../@date, time)}" xsl:validation="strict">
    <xsl:value-of select="message"/>
  </event>
</xsl:template>
```

Note:

The [xsl:merge](#) instruction is designed to enable streaming of data, so that there is no need to allocate memory to hold the input sequences. However, it can also be used in cases where streamed processing is not possible, for example when the input needs to be sorted.

15.1 Terminology for Merging

[DEFINITION: A **merge source definition** is the definition of one kind of input to the merge operation. It selects zero or more [merge input sequences](#), and it includes a [merge key specification](#) to define how the [merge key values](#) are computed for each such merge input sequence.] A merge source definition corresponds to an [xsl:merge-source](#) element in the stylesheet.

[DEFINITION: A **merge input sequence** is an arbitrary [sequence](#)^{DM30} of items which is already sorted according to the [merge key specification](#) for the corresponding [merge source definition](#).]

[DEFINITION: A **merge key specification** consists of one or more adjacent [xsl:merge-key](#) elements which together define how the [merge input sequences](#) selected by a [merge source definition](#) are sorted. Each [xsl:merge-key](#) element defines one [merge key component](#).] For example, a merge key specification for a log file might specify two merge key components, `date` and `time`.

[DEFINITION: A **merge key component** specifies one component of a [merge key specification](#); it corresponds to a single [xsl:merge-key](#) element in the stylesheet.]

[DEFINITION: For each item in a [merge input sequence](#), a value is computed for each [merge key component](#) within the [merge key specification](#). The value computed for an item by using the *N*th [merge key component](#) is referred to as the *N*th **merge key value** of that item.]

[DEFINITION: The ordered collection of [merge key values](#) computed for one item in a [merge input sequence](#) (one for each [merge key component](#) within the [merge key specification](#)) is referred to as a **composite merge key value**.]

[DEFINITION: A **merge activation** is a single evaluation of the sequence constructor contained within the [xsl:merge-action](#) element, which occurs once for each distinct [composite merge key value](#).]

15.2 The [xsl:merge](#) Instruction

```
<!-- Category: instruction -->
<xsl:merge>
  <!-- Content: (xsl:merge-source+, xsl:merge-action, xsl:fallback*) -->
</xsl:merge>
```

The effect of the [xsl:merge](#) instruction is to produce a sorted result sequence from a number of input sequences.

The input sequences to the merge operation are defined by the [xsl:merge-source](#) child elements, as described in the next section.

The sequence constructor contained in the [xsl:merge-action](#) element is evaluated once for each distinct [composite merge key value](#) to form a partial result sequence. The result of the [xsl:merge](#) instruction is the

concatenation of these partial result sequences. For example, the action might be to copy the items from all the input sequences to the result sequence without change; or it might be to select the items from one input sequence in preference to the others. In the general case, the items in the partial result sequence are produced by an arbitrary computation that has access to the items (from the various input sequences) that share the same value for the composite merge key.

The [xsl:merge-source](#) and [xsl:merge-action](#) elements are described in the following sections.

Any [xsl:fallback](#) children of the [xsl:merge](#) instruction are ignored by an XSLT 3.0 processor, but are used by an XSLT 1.0 or XSLT 2.0 processor to perform fallback processing.

Note:

An [xsl:merge](#) instruction that has no input sequences returns an empty sequence. An [xsl:merge](#) instruction with a single input sequence performs processing that is very similar in concept to [xsl:for-each-group](#) with the `group-adjacent` attribute, except that it requires the input to be sorted on the grouping key.

15.3 Selecting the Sequences to be Merged

```
<xsl:merge-source
  name? = ncname
  for-each-item? = expression
  for-each-source? = expression
  select = expression
  streamable? = boolean
  use-accumulators? = tokens
  sort-before-merge? = boolean
  validation? = "strict" | "lax" | "preserve" | "strip"
  type? = eqname >
  <!-- Content: xsl:merge-key+ -->
</xsl:merge-source>
```

Each [xsl:merge-source](#) element defines one or more [merge input sequences](#).

The `name` attribute provides a means of distinguishing items from different merge sources within the [xsl:merge-action](#) instructions. If the `name` attribute is present on an [xsl:merge-source](#) element, then it must not be equal to the `name` attribute of any sibling [xsl:merge-source](#) element. If the `name` attribute is absent, then an [implementation-dependent](#) name, different from all explicitly specified names, is allocated to the merge source.

[ERR XTSE3195] If the `for-each-item` is present then the `for-each-source`, `use-accumulators`, and `streamable` attributes must both be absent. If the `use-accumulators` attribute is present then the `for-each-source` attribute must be present. If the `for-each-source` attribute is present then the `for-each-item` attribute must be absent.

The `use-accumulators` attribute defines the set of accumulators that are applicable to the streamed document, as explained in [18.2.2 Applicability of Accumulators](#).

If neither of `for-each-item` and `for-each-source` is present, the `xsl:merge-source` element defines a single merge input sequence. This sequence is the result of evaluating the expression in the `select` attribute. This is evaluated using the dynamic context of the containing `xsl:merge` instruction. This sequence will be merged with the sequences defined by other `xsl:merge-source` elements, if present.

When the `for-each-item` attribute is present, the `xsl:merge-source` element defines a collection of merge input sequences. The selection of items in these input sequences is a two-stage process: the `for-each-item` attribute of the `xsl:merge-source` element is an expression that selects a sequence of *anchor items*, and for each anchor item, the `select` attribute is evaluated to select the items that make up one merge input sequence. The `for-each-item` expression is evaluated with the dynamic context of the containing `xsl:merge` instruction, while the `select` attribute is evaluated with the `focus` for the evaluation as follows:

- The `context item` is the anchor item
- The `context position` is the position of the anchor item within the sequence of anchor items
- The `context size` is the number of anchor items.

When the `for-each-source` attribute is present, its value must be an expression that returns a sequence of URIs. The expression is evaluated with the same dynamic context as the containing `xsl:merge` instruction. The expected type of the expression is `xs:string*`, and the actual result of the expression is converted to this type using the [function conversion rules](#). Each of these URIs is used to obtain a document node. Each **MUST** be a valid URI reference. If it is an absolute URI reference, it is used as is; if it is a relative URI reference, it is made absolute by resolving it against the base URI of the `xsl:merge-source` element. The process of obtaining a document node given a URI is the same as for the `docFO30` function, and may trigger the same error conditions. However, unlike the `docFO30` function, the `xsl:merge` instruction offers no guarantee that the resulting document will be stable (that is, that multiple calls specifying the same URI will return the same document). The resulting document nodes act as the **anchor items**. These anchor items are then used in the same way as a sequence of anchor items selected directly using the `for-each-item` attribute: in particular, the `focus` is determined in the same way.

Note:

Examples of expressions that return a sequence of URIs are:

- `for-each-source="'inputA.xml', 'inputB.xml'"`
- `for-each-source="(1 to $N) ! ('input' || $N || '.xml')"`
- `for-each-source="uri-collection('input/dir/')`

Relative URIs are resolved relative to the base URI of the `xsl:merge-source` element.

The attributes `validation` and `type` are used to control schema validation of documents read by virtue of their appearance in the result of the `for-each-source` expression. These attributes are mutually exclusive [see [ERR_XTSE1505](#)]. The rules are the same as for an `xsl:source-document` instruction specifying `streamable="yes"`. If the `for-each-source` attribute is absent, then the `validation` and `type` attributes **MUST** both be absent.

If the `sort-before-merge` attribute is absent or has the value `no`, then each input sequence **MUST** already be in the correct order for merging (a dynamic error occurs if it is not). If the attribute is present with the value `yes`, then each input sequence will first be sorted to ensure that it is in the correct order.

Example: Merging Several Documents with the Same Structure

The following `xsl:merge-source` element selects two anchor items (the root nodes of two documents), and for each of these it selects an input sequence consisting of selected `event` elements within the relevant document.

```
<xsl:merge-source for-each-source="'log-A.xml', 'log-B.xml'"  
                  streamable="yes"  
                  select="events/event">  
    <xsl:merge-key select="@timestamp" order="ascending"/>  
</xsl:merge-source>
```

This example can be extended to merge any number of input documents with the same structure:

```
<xsl:merge-source for-each-source="uri-collection('log-collection')"  
                  streamable="yes"  
                  select="events/event">  
    <xsl:merge-key select="@time" order="ascending"/>  
</xsl:merge-source>
```

In both the above examples the anchor items are document nodes, and the items in the input sequence are elements within the document that is rooted at this node. This is a common usage pattern, but by no means the only way in which the construct can be used.

The number of anchor items selected by an `xsl:merge-source` element, and therefore the number of input sequences, is variable, but the input sequences selected by one `xsl:merge-source` element must all use the same expressions to select the items in the input sequence and to compute their merge keys. If different expressions are needed for different input sequences, then multiple `xsl:merge-source` elements can be used.

Example: Merging Two Documents with Different Structure

The following code merges two log files having different internal structure:

```
<xsl:merge-source for-each-source="'event-log.xml'"  
                  streamable="yes" select="/event">  
    <xsl:merge-key select="@timestamp"/>  
</xsl:merge-source>  
<xsl:merge-source for-each-source="'error-log.xml'"  
                  streamable="yes" select="/error">  
    <xsl:merge-key select="dateTime(@date, @time)"/>  
</xsl:merge-source>
```

Although the merge keys are computed in different ways for the two input sequences, the keys must be compatible across the two sequences: in this case they are both atomic values of type `xs:dateTime`.

In the common case where there is only one input sequence of a particular kind, the `for-each-item` attribute of `xsl:merge-source` may be omitted; the `select` expression is then evaluated relative to the `focus` of the `xsl:merge` instruction itself.

Example: Sorting before Merging

Where one or more of the inputs to the merging process is not pre-sorted, a sort can be requested using the `sort-before-merge` attribute. For example:

```
<xsl:merge-source select="doc('event-log.xml')/*/event">
  <xsl:merge-key select="@timestamp"/>
</xsl:merge-source>
<xsl:merge-source select="doc('error-log.xml')//error"
  sort-before-merge="yes">
  <xsl:merge-key select="dateTime(current-date(), @time)"/>
</xsl:merge-source>
```

[ERR XTSE3190] It is a [static error](#) if two sibling `xsl:merge-source` elements have the same name.

15.4 Streamable Merging

Any input to a merging operation, provided it is selected by means of the `xsl:merge-source` element with a `for-each-source` attribute, may be designated as streamable by including the attribute `streamable="yes"` on the `xsl:merge-source` element.

When `streamable="yes"` is specified on an `xsl:merge-source` element, then (whether or not streamed processing is actually used, and whether or not the processor supports streaming) the expression appearing in the `select` attribute is implicitly used as the argument of a call on the `snapshot` function, which means that merge keys for each selected node are computed with reference to this snapshot, and the `current-merge-group` function, when used within the `xsl:merge-action` sequence constructor, delivers snapshots of the selected nodes.

Note:

There are therefore no constraints on the navigation that may be performed in computing the merge key, or in the course of evaluating the `xsl:merge-action` body. An attempt to navigate outside the portion of the source document delivered by the `snapshot` function will typically not cause an error, but will return empty results.

There is no rule to prevent the `select` expression returning atomic values, or grounded nodes from a different source document, or newly constructed nodes, but they are still processed using the `snapshot` function.

Because the `snapshot` copies accumulator values as described in [18.2.10 Copying Accumulator Values](#), the functions `accumulator-before` and `accumulator-after` may be used to gain access to information that is not directly available in the nodes that are present within each snapshot (for example, information in a header section of the merge input document).

An `xsl:merge-source` element is [guaranteed-streamable](#) if it satisfies all the following conditions:

1. The `xsl:merge-source` element has the attribute value `streamable="yes"`;
2. The `for-each-source` attribute is present on that `xsl:merge-source` element;

3. The expression in the `select` attribute of that `xsl:merge-source` element, assessed with a [context posture](#) of [striding](#) and a [context item type](#) of $U\{document-node()\}$, has [striding](#) or [grounded posture](#) and [motionless](#) or [consuming sweep](#);
4. The `sort-before-merge` attribute of that `xsl:merge-source` element is either absent or takes its default value of no.

Specifying `streamable="yes"` on an `xsl:merge-source` element declares an intent that the `xsl:merge` instruction should be streamable with respect to that particular source, either because it is [guaranteed-streamable](#), or because it takes advantage of streamability extensions offered by a particular processor. The consequences of declaring the instruction to be streamable when it is not in fact guaranteed streamable depend on the conformance level of the processor, and are explained in [19.10 Streamability Guarantees](#).

Example: Streamed Merging

The following example merges two log files, processing each of them using streaming.

```

<events>
  <xsl:merge>
    <xsl:merge-source for-each-source="'log-file-1.xml'"
                      select="/events/event"
                      streamable="yes">
      <xsl:merge-key select="@timestamp"/>
    </xsl:merge-source>
    <xsl:merge-source for-each-source="'log-files-2.xml'"
                      select="/log/day/record"
                      streamable="yes">
      <xsl:merge-key select="dateTime(../@date, time)"/>
    </xsl:merge-source>
    <xsl:merge-action>
      <events time="{current-merge-key()}">
        <xsl:copy-of select="current-merge-group()"/>
      </events>
    </xsl:merge-action>
  </xsl:merge>
</events>

```

Note that the merge key for the second merge source includes data from a child element of the selected element and also from an attribute of the parent element. This works because of the merge key is evaluated on the result of implicitly applying the [snapshot](#) function.

Example: Merging XML and non-XML Data

The following example merges two log files, one in text format and one in XML format.

```

<events>
  <xsl:merge>
    <xsl:merge-source name="fax"
      select="unparsed-text-lines('fax-log.txt')"
      <xsl:merge-key select="xs:dateTime(substring-before(., ' '))"/>
    </xsl:merge-source>
    <xsl:merge-source name="mail"
      for-each-source="'mail-log.xml'"
      select="/log/day/message"
      streamable="yes">
      <xsl:merge-key select="dateTime(../@date, @time)"/>
    </xsl:merge-source>
    <xsl:merge-action>
      <messages at="{current-merge-key()}">
        <xsl:where-populated>
          <fax>
            <xsl:for-each select="current-merge-group('fax')">
              <message xsl:expand-text="true">{
                substring-after(., ' ')
              }</message>
            </xsl:for-each>
          </fax>
          <mail>
            <xsl:sequence select="current-merge-group('mail')/*"/>
          </mail>
        </xsl:where-populated>
      </messages>
    </xsl:merge-action>
  </xsl:merge>
</events>
```

15.5 Defining the Merge Keys

The keys on which the input sequences are sorted are referred to as merge keys. If the attribute `sort-before-merge` has the value `yes`, the input sequences will be sorted into the correct sequence before the merge operation takes place (alternatively, the processor MAY use an algorithm that has the same effect as sorting followed by merging). If the attribute is absent or has the value `no`, then the input sequences MUST already be in the correct order.

The merge key for each type of input sequence (that is, for each `xsl:merge-source` element) is defined by a sequence of `xsl:merge-key` element children of the `xsl:merge-source` element. Each `xsl:merge-key` element defines one merge key component. The syntax and semantics of an `xsl:merge-key` element are closely based on the rules for the `xsl:sort` element (the only exception being the absence of the `stable` attribute); the difference is that `xsl:merge-key` elements do not cause a sort to take place, they merely declare the existing sort order of the input sequence.

```

<xsl:merge-key
  select? = expression
  lang? = { language }
  order? = { "ascending" | "descending" }
  collation? = { uri }
  case-order? = { "upper-first" | "lower-first" }
  data-type? = { "text" | "number" | eqname } >
  <!-- Content: sequence-constructor -->
</xsl:merge-key>
```

The `select` attribute and the contained [sequence constructor](#) are mutually exclusive:

[ERR XTSE3200] It is a [static error](#) if an `xsl:merge-key` element with a `select` attribute has non-empty content.

The value of N th merge key value of an item J in a [merge input sequence](#) S is the result of the expression in the `select` attribute of the N th `xsl:merge-key` child of the corresponding `xsl:merge-source` element, or in the absence of the `select` attribute, the result of the contained [sequence constructor](#). This is evaluated with a [singleton focus](#) based on J , or, if `streamable=yes` is specified on the `xsl:merge-source`, a [singleton focus](#) based on a [snapshot](#) of J (see [15.4 Streamable Merging](#)).

Note:

This means that `position()` and `last()` return 1 (one). This differs from the way `xsl:sort` keys are evaluated, where `position()` is the position in the unsorted sequence, and `last()` is the size of the unsorted sequence.

The effect of the `xsl:merge-key` elements is defined in terms of the rules for an equivalent sequence of `xsl:sort` elements: if the rules for sorting (see [13.1.1 The Sorting Process](#)) with `stable="yes"` would place an item A before an item B in the [sorted sequence](#) produced by the sorting process, then A must precede B in the input sequence to the merging process.

The merge keys of the various input sequences to a merge operation must be compatible with each other, since the merge operation will decide the ordering of the result sequence by comparing merge key values across input sequences. This means that across all the `xsl:merge-source` children of an `xsl:merge` instruction:

- Each `xsl:merge-source` element MUST have the same number of `xsl:merge-key` child elements; let this number be N .
- For each integer J in $1..N$, consider the set of `xsl:merge-key` elements that are in position J among the `xsl:merge-key` children of their parent `xsl:merge-source` element. All the `xsl:merge-key` elements in this set MUST have the same [effective value](#) for their `lang`, `order`, `collation`, `case-order`, and `data-type` attributes, where having the same effective value in this case means that either both attributes must be absent, or both must be present and evaluate to the same value; and in addition in the case of `collation` the absolute URI must be the same after resolving against the base URI.

If any of the attributes `lang`, `order`, `collation`, `case-order`, or `data-type` are [attribute value templates](#), then their [effective values](#) are evaluated using the [focus](#) of the containing `xsl:merge` instruction.

[ERR XTSE2200] It is a [static error](#) if the number of `xsl:merge-key` children of a `xsl:merge-source` element is not equal to the number of `xsl:merge-key` children of another `xsl:merge-source` child of the same

[xsl:merge](#) instruction.

[ERR XTDE2210] It is a [dynamic error](#) if there are two [xsl:merge-key](#) elements that occupy corresponding positions among the [xsl:merge-key](#) children of two different [xsl:merge-source](#) elements and that have differing [effective values](#) for any of the attributes `lang`, `order`, `collation`, `case-order`, or `data-type`. Values are considered to differ if the attribute is present on one element and not on the other, or if it is present on both elements with [effective values](#) that are not equal to each other. In the case of the `collation` attribute, the values are compared as absolute URIs after resolving against the base URI. The error `MAY` be reported statically if it is detected statically.

[ERR XTDE2220] It is a [dynamic error](#) if any input sequence to an [xsl:merge](#) instruction contains two items that are not correctly sorted according to the merge key values defined on the [xsl:merge-key](#) children of the corresponding [xsl:merge-source](#) element, when compared using the collation rules defined by the attributes of the corresponding [xsl:merge-key](#) children of the [xsl:merge](#) instruction, unless the attribute `sort-before-merge` is present with the value `yes`.

[ERR XTTE2230] It is a [type error](#) if some item selected by a particular merge key in one input sequence is not comparable using the XPath `le` operator with some item selected by the corresponding sort key in another input sequence.

[15.6 The Current Merge Group and Key](#)

During processing of an [xsl:merge](#) instruction, two additional values are available within the dynamic context:

- [DEFINITION: The **current merge group** is a [map](#). During evaluation of an [xsl:merge](#) instruction, as each group of items with equal [composite merge key values](#) is processed, the current merge group is set to a map whose keys are the names of the various merge sources, and whose associated values are the items from each merge source having the relevant composite merge key value.]
- [DEFINITION: The **current merge key** is a sequence of atomic values. During evaluation of an [xsl:merge](#) instruction, as each group of items with equal [composite merge key values](#) is processed, the current merge key is set to the composite merge key value that these items have in common.]

These values are made available through the functions [current-merge-group](#) and [current-merge-key](#).

The [current merge group](#) and [current merge key](#) are available within the sequence constructor contained by an [xsl:merge-action](#) element. The values are initially [absent](#) during the evaluation of global variables and stylesheet parameters, during the evaluation of the `use` attribute or contained sequence constructor of [xsl:key](#), and during the evaluation of the `initial-value` attribute of [xsl:accumulator](#) and the `select` attribute of contained sequence constructor of [xsl:accumulator-rule](#). All [invocation constructs](#) set the [current merge group](#) and [current merge key](#) to [absent](#).

Note:

Taken together, these rules mean that any invocation of [current-merge-group](#) or [current-merge-key](#) that is not lexically scoped by an [xsl:merge-action](#) element will raise a dynamic error.

When an inner [xsl:merge](#) instruction is lexically nested within the [xsl:merge-action](#) element of an outer [xsl:merge](#) instruction, any use of [current-merge-group](#) or [current-merge-key](#) that appears within the

`xsl:merge-action` of the inner `xsl:merge` instruction is a reference to the `current merge group` or `current merge key` of the inner `xsl:merge` instruction, while any such reference that appears within the outer `xsl:merge-action` element, but not within the inner `xsl:merge-action`, is a reference to the `current merge group` or `current merge key` of the outer `xsl:merge` instruction. This means, for example, that a reference to the current merge group of the outer `xsl:merge` can appear in the `select` attribute of an `xsl:merge-source` child of the inner `xsl:merge`.

On completion of the evaluation of the `xsl:merge-action` sequence constructor, the current merge group and current merge key revert to their previous values.

15.6.1 `fn:current-merge-group`

Summary

Returns the group of items currently being processed by an `xsl:merge` instruction.

Signatures

```
fn:current-merge-group() as item()*
```

```
fn:current-merge-group($source as xs:string) as item()*
```

Properties

This function is `deterministic`^{FO30}, `context-dependent`^{FO30}, and `focus-independent`^{FO30}.

Rules

The `current merge group` is bound during evaluation of the `xsl:merge-action` child of an `xsl:merge` instruction. If no `xsl:merge-action` is being evaluated, then the current merge group is `absent`, in which case the function raises a dynamic error (see below).

The `current merge group` (if not absent) is a `map`. It contains the set of items, from all merge inputs, that share a common value for the merge key. This is structured as a map so that the items from each merge source can be identified. The key in the map is the value of the `name` attribute of the corresponding `xsl:merge-source` element (or an invented name, in its absence), and the associated value is the set of items contributed by that merge group.

The map itself is not made visible, but this function returns values derived from the map. Specifically, if the map is denoted by `$G`:

- The single-argument form of this function returns the value of the expression `if (map:contains($source)) then $G($source) else error()`. Informally, if there is an `xsl:merge-source` element whose `name` attribute matches `$source`, the function returns the items in the current merge group that are contributed by this merge source; otherwise it raises a dynamic error (see below).
- The zero-argument form of the function returns the value of the expression `sort(map:keys($G)) !$G(.)`, where the `sort()` function sorts the names of `xsl:merge-source` elements into the document order of the `xsl:merge-source` elements in the stylesheet. Informally, it returns all the items in the current merge group regardless of which merge source they derive from.

Within the `current merge group`, the ordering of items from the input sequences is as follows, in major-to-minor order:

- Items are first ordered by the `xsl:merge-source` element that defined the input sequence from which the item was taken; items from `xsl:merge-source A` precede items from `xsl:merge-source B` if *A* precedes *B* in document order within the stylesheet.
- Items from different input sequences selected by the same `xsl:merge-source` element are then ordered based on the order of the anchor items in the sequence selected by evaluating the `select` attribute of the `xsl:merge-source` element.
- Finally, duplicate items from the same input sequence retain their order from the input sequence.

Duplicates are not eliminated: for example, if the same node is selected in more than one input sequence, it may appear twice in the current merge group.

Error Conditions

[ERR XTSE3470] It is a [static error](#) if the `current-merge-group` function is used within a [pattern](#).

[ERR XTDE3480] It is a [dynamic error](#) if the `current-merge-group` function is used when the current merge group is [absent](#). The error [MAY](#) be reported statically if it can be detected statically.

[ERR XTDE3490] It is a [dynamic error](#) if the `$source` argument of the `current-merge-group` function does not match the `name` attribute of any `xsl:merge-source` element for the current merge operation. The error [MAY](#) be reported statically if it can be detected statically.

Notes

Because the `current-merge-group` is cleared by function calls and template calls, the `current-merge-group` function only has useful effect when the call appears as a descendant of an `xsl:merge-action` element.

If an `xsl:merge-source` element has no `name` attribute, then it is not possible to discover the items in the current merge group that derive specifically from that source, but these items will still be present in the current merge group, and will be included in the result when the function is called with no arguments.

Like other XSLT extensions to the dynamic evaluation context, the `current-merge-group` is not retained as part of the closure of a function value. This means that the expression `current-merge-group#0` is valid and returns a function value, but any invocation of this function will fail with a dynamic error [see [ERR XTDE3480](#)].

15.6.2 `fn:current-merge-key`

Summary

Returns the merge key of the merge group currently being processed using the `xsl:merge` instruction.

Signature

```
fn:current-merge-key() as xs:anyAtomicType*
```

Properties

This function is [deterministic^{FO30}](#), [context-dependent^{FO30}](#), and [focus-independent^{FO30}](#).

Rules

The evaluation context for XPath [expressions](#) includes a component called the [current merge key](#), which is a sequence of atomic values. The current merge key is the [composite merge key value](#) shared in common by all the items within the [current merge group](#).

The function [current-merge-key](#) returns the [current merge key](#).

While the [xsl:merge-action](#) child of an [xsl:merge](#) instruction is being evaluated, the [current merge key](#) will be a single atomic value if there is a single merge key, or a sequence of atomic values if there are multiple merge keys.

At other times, the current merge key will be [absent](#).

The [merge keys](#) of all items in a group are not necessarily identical. For example, one might be an [xs:float](#) while another is a numerically equal [xs:decimal](#). The [current-merge-key](#) function returns the merge key of the first item in the group, after atomization and casting of [xs:untypedAtomic](#) values to [xs:string](#).

Error Conditions

[ERR XTSE3500] It is a [static error](#) if the [current-merge-key](#) function is used within a [pattern](#).

[ERR XTDE3510] It is a [dynamic error](#) if the [current-merge-key](#) function is used when the current merge key is [absent](#), or when it is invoked in the course of evaluating a pattern. The error [MAY](#) be reported statically if it can be detected statically.

Notes

Like other XSLT extensions to the dynamic evaluation context, the [current merge key](#) is not retained as part of the closure of a function value. This means that the expression `current-merge-key#0` is valid and returns a function value, but any invocation of this function will fail with a dynamic error [see [ERR XTDE3510](#)].

15.7 The [xsl:merge-action](#) Element

The [xsl:merge-action](#) child of an [xsl:merge](#) instruction defines the processing to be applied for each distinct [composite merge key value](#) found in the input sequences to the [xsl:merge](#) instruction.

```
<xsl:merge-action>
  <!-- Content: sequence-constructor -->
</xsl:merge-action>
```

The merge key values for each item in an input sequence are calculated based on the corresponding [xsl:merge-key](#) elements, in the same way as [sort key values](#) are calculated using a sequence of [xsl:sort](#) elements (see [13.1.1 The Sorting Process](#)). If several items from the same or from different input sequences have the same values for all their merge keys (comparing pairwise), then they are considered to form a group. The sequence constructor contained in the [xsl:merge-action](#) element is evaluated once for each such group of items, and the result of the [xsl:merge](#) instruction is the concatenation of the results obtained by processing each group in turn.

The groups are processed one by one, based on the values of the merge keys for the group. If group G has a set of merge key values M , while group H has a set of merge key values N , then in the result of the [xsl:merge](#) instruction, the result of processing group G will precede the result of processing H if and only if M precedes N in

the sort order defined by the `lang`, `order`, `collation`, `case-order`, and `data-type` attributes of the merge key definitions.

Generally, two sets of merge key values are distinct if any corresponding items in the two sets of values do not compare equal under the rules for the XPath `eq` operator, under the collating rules for the corresponding merge key definition. In rare cases, when considering more than two sets of merge key values, ambiguities may arise because of the non-transitivity of the `eq` operator when applied across different numeric types. In this situation, the partitioning of items into sets having distinct key values is handled in the same way as for [`xsl:for-each-group`](#) (see [14.5 Non-Transitivity](#)), and is to some extent [implementation-dependent](#).

The [focus](#) for evaluation of the sequence constructor contained in the [`xsl:merge-action`](#) element is as follows:

- The [context item](#) is the first item in the group being processed, that is `current-merge-group()[1]`
- The [context position](#) is the position of the current group within the sequence of groups (so the first evaluation of [`xsl:merge-action`](#) has `position()=1`, the second has `position()=2`, and so on).
- The [context size](#) is as follows:
 - If any of the [`xsl:merge-source`](#) elements within the [`xsl:merge`](#) instruction specifies `streamable="yes"` (explicitly or implicitly), then absent.

Note:

This means that within the [`xsl:merge-action`](#) of a streamable [`xsl:merge`](#), calling `last()` throws error [\[ERR_XPDY0002\]](#)^{XP30}.

- Otherwise, the number of groups, that is, the number of distinct sets of merge key values.

Example: Selective Processing of Merge Inputs

Consider a situation where there are two merge sources, named "master" and "update"; the master source identifies a single merge input file (the master file), while the update source identifies a set of N update files, perhaps one for each day of the week. The required logic is that if a merge key is present only in the master file, then the corresponding item should be copied to the output; if it is present in a single update file then that item replaces the corresponding item from the master file; if it is present in several update files, then an error is raised. This can be achieved as follows:

```

<xsl:merge>
  <xsl:merge-source name="master"
    for-each-source="master.xml"
    streamable="yes"
    select="/events/event">
    <xsl:merge-key select="@key"/>
  </xsl:merge-source>
  <xsl:merge-source name="updates"
    for-each-source="uri-collection('updates')"
    streamable="yes"
    select="/events/event-change">
    <xsl:merge-key select="@affected-key"/>
  </xsl:merge-source>
  <xsl:merge-action>
    <xsl:choose>
      <xsl:when test="empty(current-merge-group('master'))">
        <xsl:message>
          Error: update is present with no matching master record!
        </xsl:message>
      </xsl:when>
      <xsl:when test="empty(current-merge-group('updates'))">
        <xsl:copy-of select="current-merge-group('master')"/>
      </xsl:when>
      <xsl:when test="count(current-merge-group('updates')) = 1">
        <xsl:copy-of select="current-merge-group('updates')"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:message>
          Conflict: multiple updates for the same master record!
        </xsl:message>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:merge-action>
</xsl:merge>
```

Some words of explanation:

- Error messages are produced if there is an update element whose key does not correspond to any element in the master source, or if there is more than one update element corresponding to the same master element.
- In the absence of errors, if there is a single update element then it is copied to the output; if there is none, then the master element is copied.

15.8 Examples of `xsl:merge`

Previous sections introduced examples designed to illustrate some specific features of the `xsl:merge` instruction. This section provides some further examples to illustrate different ways in which the instruction can be used.

Example: Applying Transactions to a Master File

This example applies transactions from a transaction file to a master file. Records in the master file for which there is no corresponding transaction are copied unchanged. The transaction file contains instructions to delete, replace, or insert records identified by an ID value. The master file is known to be sorted on the ID value; the transaction file is unsorted.

Master file document structure:

```
<data>
  <record ID="A0001">...</record>
  <record ID="A0002">...</record>
  <record ID="A0003">...</record>
</data>
```

Transaction file document structure:

```
<transactions>
  <update record="A0004" action="insert">...</update>
  <update record="A0002" action="delete"/>
  <update record="A0003" action="replace">...</update>
</transactions>
```

Solution:

```
<xsl:merge>
  <xsl:merge-source name="master"
    select="doc('master.xml')/data/record">
    <xsl:merge-key select="@ID"/>
  </xsl:merge-source>
  <xsl:merge-source name="updates"
    sort-before-merge="yes"
    select="doc('transactions.xml')/transactions/update">
    <xsl:merge-key select="@record"/>
  </xsl:merge-source>
  <xsl:merge-action>
    <xsl:choose>
      <xsl:when test="empty(current-merge-group('updates'))">
        <xsl:copy-of select="current-merge-group('master')"/>
      </xsl:when>
      <xsl:when test="current-merge-group('updates')/@action=('insert',
'replace')">
        <record ID="{current-merge-key()}">
          <xsl:copy-of select="current-merge-group('updates')/*"/>
        </record>
      </xsl:when>
      <xsl:when test="current-merge-group('updates')/@action='delete'">
        </xsl:choose>
    </xsl:merge-action>
  </xsl:merge>
```

Example: Merging Two Sequences of Numbers

The [xsl:merge](#) instruction can be used to determine the union, intersection, or difference of two sequences of numbers (or other atomic values). This code gives the union:

```
<xsl:merge>
  <xsl:merge-source select="1 to 30">
    <xsl:merge-key select=". "/>
  </xsl:merge-source>
  <xsl:merge-source select="20 to 40">
    <xsl:merge-key select=". "/>
  </xsl:merge-source>
  <xsl:merge-action>
    <xsl:sequence select="current-merge-key()"/>
  </xsl:merge-action>
</xsl:merge>
```

While this gives the intersection:

```
<xsl:merge>
  <xsl:merge-source select="1 to 30">
    <xsl:merge-key select=". "/>
  </xsl:merge-source>
  <xsl:merge-source select="20 to 40">
    <xsl:merge-key select=". "/>
  </xsl:merge-source>
  <xsl:merge-action>
    <xsl:if test="count(current-merge-group()) eq 2">
      <xsl:sequence select="current-merge-key()"/>
    </xsl:if>
  </xsl:merge-action>
</xsl:merge>
```

16 Splitting

Sometimes it is convenient to be able to compute multiple results during a single scan of the input data. For example, a transformation may wish to rename selected elements, and also to output a count of how many elements have been renamed. Traditionally in a functional language this means computing two separate functions of the input sequence, which (in the absence of sophisticated optimization) will result in the input being scanned twice. This is inconsistent with streaming, where the input is only available to be scanned once, and it can also lead to poor performance in non-streaming applications.

To meet this requirement, XSLT 3.0 introduces the instruction [xsl:fork](#). The content of this instruction is a restricted form of [sequence constructor](#), and in a formal sense the effect of the instruction is simply to return the result of evaluating the sequence constructor. However, the presence of the instruction affects the analysis of streamability (see [19 Streamability](#)). In particular, when [xsl:fork](#) is used in a context where streaming is required, each independent instruction within the sequence constructor must be streamable, but the analysis assumes that these instructions can all be evaluated during a single pass of the streamed input document.

Note:

The semantics of the instruction require a number of result sequences to be computed during a single pass of the input. A processor may interpret this as a request to use multiple threads. However, implementations using a single thread are feasible, and this instruction is not intended primarily as a means for stylesheet authors to express their intentions with regard to multi-threaded execution.

Note:

Because multiple results are computed during a single pass of the input, and then concatenated into a single sequence, this instruction will generally involve some buffering of results. The amount of memory used should not exceed that needed to hold the results of the instruction. However, within this principle, implementations may adopt a variety of strategies for evaluation; for example, there may be cases where buffering of the input is more efficient than buffering of output.

Generally, stylesheet authors indicate that buffering of input is the preferred strategy by using the [copy-of](#) or [snapshot](#) functions, and indicate that buffering of output is preferred by using [xsl:fork](#). However, conformant processors are not constrained in their choice of evaluation strategies.

The content model of the [xsl:fork](#) instruction (given that an XSLT 3.0 processor ignores [xsl:fallback](#)) takes two possible forms:

1. A sequence of [xsl:sequence](#) instructions
2. A single [xsl:for-each-group](#) instruction. This will normally use the group-by attribute, because in all other cases the containing [xsl:fork](#) instruction has no useful effect.

The first form is appropriate when splitting a single input stream into a fixed number of output streams, known statically: for example, one output stream for credit transactions, a second for debit transactions. The second form is appropriate when the number of output streams depends on the data: for example, one output stream for each distinct city name found in the input data.

The following section describes the [xsl:fork](#) instruction more formally.

16.1 [The xsl:fork Instruction](#)

```
<!-- Category: instruction -->
<xsl:fork>
  <!-- Content: (<xsl:fallback*>, (<xsl:sequence>, <xsl:fallback*>)* | (<xsl:for-each-
group>, <xsl:fallback*>)) -->
</xsl:fork>
```

Note:

The content model can be described as follows: there is either a single [xsl:for-each-group](#) instruction, or a sequence of zero or more [xsl:sequence](#) instructions; in addition, [xsl:fallback](#) instructions may be added anywhere.

The result of the [xsl:fork](#) instruction is the sequence formed by concatenating the results of evaluating each of its contained instructions, in order. That is, the result can be determined by treating the content as a [sequence constructor](#) and evaluating it as such.

Note:

Any [xsl:fallback](#) children will be ignored by an XSLT 3.0 processor.

By using the [xsl:fork](#) instruction, the stylesheet author is suggesting to the [processor](#) that buffering of output is acceptable even though this might use unbounded memory and thus violate the normal expectations of streamable processing.

The presence of an [xsl:fork](#) instruction affects the analysis of streamability, as described in [19 Streamability](#).

[16.2 Examples of Splitting with Streamed Data](#)

This section gives examples of how splitting using [xsl:fork](#) can be used to enable streaming of input documents in cases where several results need to be computed during a single pass over the input data.

Example: Splitting a Transaction File into Credits and Debits

Consider a transaction file that contains a sequence of debits and credits:

```
<transactions>
  <transaction value="5.60"/>
  <transaction value="11.20"/>
  <transaction value="-3.40"/>
  <transaction value="8.90"/>
  <transaction value="-1.99"/>
</transactions>
```

where the requirement is to split this into two separate files containing credits and debits respectively.

This can be achieved in guaranteed-streamable code as follows:

```
<xsl:source-document streamable="yes" href="transactions.xml">
  <xsl:fork>
    <xsl:sequence>
      <xsl:result-document href="credits.xml">
        <credits>
          <xsl:for-each select="transactions/transaction[@value &gt;= 0]">
            <xsl:copy-of select=". />
          </xsl:for-each>
        </credits>
      </xsl:result-document>
    </xsl:sequence>
    <xsl:sequence>
      <xsl:result-document href="debits.xml">
        <debits>
          <xsl:for-each select="transactions/transaction[@value &lt; 0]">
            <xsl:copy-of select=". />
          </xsl:for-each>
        </debits>
      </xsl:result-document>
    </xsl:sequence>
  </xsl:fork>
</xsl:source-document>
```

In the absence of the `xsl:fork` instruction, this would not be streamable, because the sequence constructor includes two consuming instructions. With the addition of the `xsl:fork` instruction, however, each `xsl:result-document` instruction is allowed to make a downwards selection.

One possible implementation model for this is as follows: a single thread reads the source document, and sends parsing events such as start-element and end-element to two other threads, each of which is writing one of the two result documents. Each of these implements the downwards-selecting path expression using a process that waits until the next `transaction` start-element event is received; when this event is received, the process examines the `@value` attribute to determine whether or not this transaction is to be copied; if it is, then all events until the matching `transaction` end-element event are copied to the serializer for the result document; otherwise, these events are discarded.

Example: Splitting a Transaction File by Customer Account

Consider a transaction file that contains a sequence of debits and credits:

```
<transactions>
  <transaction value="5.60" account="01826370"/>
  <transaction value="11.20" account="92741838"/>
  <transaction value="-3.40" account="01826370"/>
  <transaction value="8.90" account="92741838"/>
  <transaction value="-1.99" account="43861562"/>
</transactions>
```

where the requirement is to split this into a number of separate files, one for each account number found in the input.

This can be achieved in guaranteed-streamable code as follows:

```
<xsl:source-document streamable="yes" href="transactions.xml">
  <xsl:fork>
    <xsl:for-each-group select="transactions/transaction" group-by="@account">
      <xsl:result-document href="account{current-grouping-key()}.xml">
        <transactions account="{current-grouping-key()}>
          <xsl:copy-of select="current-group()"/>
        </transactions>
      </xsl:result-document>
    </xsl:for-each-group>
  </xsl:fork>
</xsl:source-document>
```

In the absence of the `xsl:fork` instruction, this would not be streamable, because in the general case the output of `xsl:for-each-group` with a `group-by` attribute needs to be buffered. (The streamability rules do not recognize an `xsl:for-each-group` whose body comprises an `xsl:result-document` instruction as a special case.) With the addition of the `xsl:fork` instruction, however, the code becomes guaranteed streamable.

One possible implementation model for this is as follows: the processor opens a new serializer each time a new account number is encountered in the input, and writes the `<transactions>` start tag to the serializer. When a `transaction` element is encountered in the input, it is copied to the relevant serializer, according to the value of the `account` attribute. At the end of the input, a `<transactions>` end tag is written to each of the serializers, and each output file is closed.

In the more general case, where the body of the `xsl:for-each-group` instruction contributes output to the principal result document, the output generated by processing each group needs to be buffered in memory. The requirement to use `xsl:fork` exists so that this use of (potentially unbounded) memory has to be a conscious decision by the stylesheet author.

Example: Arithmetic using Multiple Child Elements as Operands

The rules for streamability do not allow two instructions in a sequence constructor to both read child or descendant elements of the context node, which makes it tricky to perform a calculation in which multiple child elements act as operands. This restriction can be avoided by using `xsl:fork`, as shown below, where each of the two branches of the `xsl:fork` instruction selects children of the context node.

```
<xsl:template match="order" mode="a-streamable-mode">
  <xsl:variable name="price-and-discount" as="xs:decimal+">
    <xsl:fork>
      <xsl:sequence select="xs:decimal(price)" />
      <xsl:sequence select="xs:decimal(discount)" />
    </xsl:fork>
  </xsl:variable>
  <xsl:value-of select="$price-and-discount[1] - $price-and-discount[2]" />
</xsl:template>
```

A possible implementation strategy here is for events from the XML parser to be sent to two separate agents (perhaps but not necessarily running in different threads), one of which computes `xs:decimal(price)` and the other `xs:decimal(discount)`; on completion the results computed by the two agents are appended to the sequence that forms the value of the variable.

With this strategy, the processor would require sufficient memory to hold the results of evaluating each branch of the fork. If these results (unlike this example) are large, this could defeat the purpose of streaming by requiring large amounts of memory; nevertheless, this code is treated as streamable.

Note:

An alternative solution to this requirement is to use map constructors: see [21.4 Map Constructors](#).

Example: Deleting Elements, and Counting Deletions

In this example the input is a narrative document containing `note` elements at any level of nesting. The requirement is to output a copy of the input document in which (a) the `note` elements have been removed, and (b) a `footnote` is added at the end indicating how many `note` elements have been deleted.

```
<xsl:mode on-no-match="shallow-copy" streamable="yes"/>

<xsl:template match="note"/>

<xsl:template match="/*">
  <xsl:fork>
    <xsl:sequence>
      <xsl:apply-templates/>
    </xsl:sequence>
    <xsl:sequence>
      <footnote>
        <p>Removed <xsl:value-of select="count(./note)" />
           note elements.</p>
      </footnote>
    </xsl:sequence>
  </xsl:fork>
</xsl:template>
```

The `xsl:fork` instruction contains two independent branches. These can therefore be evaluated in the same pass over the input data. The first branch (the `xsl:apply-templates` instruction) causes everything except the `note` elements to be copied to the result; the second instruction (the literal result element `footnote`) outputs a count of the number of descendant `note` elements.

Note that although the processing makes a single pass over the input stream, there is some buffering of results required, because the results of the instructions within the `xsl:fork` instruction need to be concatenated. In this case an intelligent implementation might be able to restrict the buffered data to a single integer.

In a formal sense, however, the result is exactly the same as if the `xsl:fork` element were not there.

An alternative way of solving this example problem would be to count the number of `note` elements using an accumulator: see [18.2 Accumulators](#).

17 Regular Expressions

The function library for XPath 3.0 defines several functions that make use of regular expressions:

- [matches](#)^{F030} returns a boolean result that indicates whether or not a string matches a given regular expression.
- [replace](#)^{F030} 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](#)^{F030} returns a sequence of strings formed by breaking a supplied input string at any separator that matches a given regular expression.

- [analyze-string](#)^{FO30} returns a tree of nodes that effectively add markup to a string indicating the parts of the string that matched the regular expression, as well as its captured groups.

These functions are described in [\[Functions and Operators 3.0\]](#).

Supplementing these functions, XSLT provides an instruction [xsl:analyze-string](#), which is defined in this section.

Note:

The [xsl:analyze-string](#) instruction predates the [analyze-string](#)^{FO30} function, and provides very similar functionality, though in a different way. The two constructs are not precisely equivalent; for example, [xsl:analyze-string](#) allows a regular expression that matches a zero-length string while the [analyze-string](#)^{FO30} function does not. The [xsl:analyze-string](#) instruction (via the use of [regex-group](#)) provides information about the value of captured substrings; the [analyze-string](#)^{FO30} function additionally provides information about the position of the captured substrings within the original string.

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 [\[Functions and Operators 3.0\]](#) (see [Section 5.6.1 Regular expression syntax](#)^{FO30}), which is itself based on the syntax defined in [\[XML Schema Part 2\]](#).

17.1 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 an empty sequence, it is treated as a zero-length string. If the value 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`^{FO30}, `replace`^{FO30}, and `tokenize`^{FO30}. 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 [\[Functions and Operators 3.0\]](#) ([Section 5.6.1.1 Flags](#)^{FO30}).

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 `xsl:analyze-string` instruction may have two child elements: `xsl:matching-substring` and `xsl:non-matching-substring`. Both elements are optional, and neither may appear more than once. At least one of them must be present. If both are present, the `xsl:matching-substring` element must come first.

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 or 3.0 processor, but allow fallback behavior to be defined when the stylesheet is used with an XSLT 1.0 processor operating with forwards-compatible behavior.

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 [dynamic error](#) if the [effective value](#) of the `regex` attribute does not conform to the REQUIRED syntax for regular expressions, as specified in [\[Functions and Operators 3.0\]](#). 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 [dynamic error](#) if the [effective value](#) of the `flags` attribute has a value other than the values defined in [\[Functions and Operators 3.0\]](#). 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](#).

To explain the behavior of the instruction it is useful to consider an input string of length N characters as having $N+1$ inter-character positions, including one just before the first character and one just after the last. Each of these

positions is a possible position for testing whether the regular expression matches. These positions are numbered from zero to N.

Note:

The term **character**, here as elsewhere in this specification, means a Unicode codepoint. When strings are held in decomposed form, the multiple codepoints representing a composite character are considered to be multiple characters. A codepoint greater than 65535 is considered as one character, not as a surrogate pair.

The processor starts by setting the current position to position zero, and the current non-matching substring to a zero-length string. It then does the following repeatedly:

1. Test whether the regular expression matches at the current position.
2. If there is a match:
 - a. If the current non-matching substring has length greater than zero, evaluate the [`xsl:non-matching-substring`](#) sequence constructor with the current non-matching substring as the context item.
 - b. Reset the current non-matching substring to a zero-length string.
 - c. Evaluate the [`xsl:matching-substring`](#) sequence constructor with the matching substring as the context item.
 - d. Do the appropriate one of the following:
 - i. If the matching substring is non-zero length, set the current position to coincide with the end of the matching substring, exit, and repeat.
 - ii. If the matching substring is zero length and the current position is at the end of the input string, exit.
 - iii. If the matching substring is zero length and the current position is not at the end of the input string, add the character that immediately follows the current position to the current non-matching substring, set the current position to the position immediately after this character, exit, and repeat.
3. If there is no match:
 - a. If the current position is the last position (that is, just after the last character):
 - i. If the current non-matching substring has length greater than zero, evaluate the [`xsl:non-matching-substring`](#) sequence constructor with the current non-matching substring as the context item.
 - ii. Exit.
 - b. Otherwise, add the character at the current position to the current non-matching substring, increment the current position, and repeat.

When the matcher is looking for a match at a particular starting position and there are several alternatives within the regular expression that match at this 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`.

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 non-matching substring will contain at least one character, but a matching substring may be zero-length. This sequence of substrings is processed using the instructions within the contained [`xsl:matching-substring`](#) and [`xsl:non-matching-substring`](#) elements. 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](#).

17.2 [fn:regex-group](#)

Summary

Returns the string captured by a parenthesized subexpression of the regular expression used during evaluation of the [xsl:analyze-string](#) instruction.

Signature

```
fn:regex-group($group-number as xs:integer) as xs:string
```

Properties

This function is [deterministic^{FO30}](#), [context-dependent^{FO30}](#), and [focus-independent^{FO30}](#).

Rules

[DEFINITION: While the [xsl:matching-substring](#) instruction is active, a set of **current captured substrings** is available, corresponding to the parenthesized subexpressions 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, excluding any non-capturing groups, which are written as `(?:xxx)`. The zeroth 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 subexpression with the given number.
3. The parenthesized subexpression exists, and did not match any part of the input string.
4. The parenthesized subexpression 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](#).

17.3 [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.

Example: Matching Zero-Length Strings

This example removes all empty and whitespace-only lines from a file.

```
<xsl:analyze-string select="unparsed-text('in.txt')"
    regex="^[\t]*$" flags="m" expand-text="yes">
    <xsl:non-matching-substring>{.}</xsl:non-matching-substring>
</xsl:analyze-string>
```

Example: Parsing comma-separated values

There are many variants of CSV formats. This example is designed to handle input where:

- Each record occupies one line.
- Fields are separated by commas.
- Quotation marks around a field are optional, unless the field contains a comma or quotation mark, in which case they are mandatory.
- A quotation mark within the value of a field is represented by a pair of two adjacent quotation marks.

For example, the input record:

```
Ten Thousand,10000,, "10,000", "It's ""10 Grand""", mister",10K
```

contains six fields, specifically:

- Ten Thousand
- 10000
- <zero-length-string>
- 10,000
- It's "10 Grand", mister
- 10K

The following code parses such CSV input into an XML structure containing `row` and `col` elements:

```
<xsl:for-each select="unparsed-text-lines('in.csv')" expand-text="yes">
  <row>
    <xsl:analyze-string select="."
      regex='(?:^|,)((?:([":"]|"")*)|(("[^",]*))'>
      <xsl:matching-substring>
        <col>{replace(regex-group(1), '""', '') || regex-group(2)}</col>
      </xsl:matching-substring>
    </xsl:analyze-string>
  </row>
</xsl:for-each>
```

Note that because this regular expression matches a zero-length string, it is not permitted in XSLT 2.0.

18 Streaming

XSLT 3.0 introduces a number of constructs that are specifically designed to enable streamed applications to be written, but which are also useful in their own right; it also includes some features that are very specialized to streaming.

18.1 The `xsl:source-document` Instruction

```
<!-- Category: instruction -->
<xsl:source-document
  href = { uri }
  streamable? = boolean
  use-accumulators? = tokens
  validation? = "strict" | "lax" | "preserve" | "strip"
  type? = eqname >
<!-- Content: sequence-constructor -->
</xsl:source-document>
```

The [xsl:source-document](#) instruction reads a source document whose URI is supplied, and processes the content of the document by evaluating the contained [sequence constructor](#). The `streamable` attribute (default "no") allows streamed processing to be requested.

For example, if a document represents a book holding a sequence of chapters, then the following code can be used to split the book into multiple XML files, one per chapter, without allocating memory to hold the entire book in memory at one time:

```
<xsl:source-document streamable="yes" href="book.xml">
  <xsl:for-each select="book">
    <xsl:for-each select="chapter">
      <xsl:result-document href="chapter{position()}.xml">
        <xsl:copy-of select=". />
      </xsl:result-document>
    </xsl:for-each>
  </xsl:for-each>
</xsl:source-document>
```

Note:

In earlier drafts of this specification the [xsl:source-document](#) element was named [xsl:stream](#). The instruction has been generalised to handle both streamed and unstreamed input.

The document to be read is determined by the [effective value](#) of the `href` attribute (which is defined as an [attribute value template](#)). This must be a valid URI reference. If it is an absolute URI reference, it is used as is; if it is a relative URI reference, it is made absolute by resolving it against the base URI of the [xsl:source-document](#) element. The process of obtaining a document node given a URI is the same as for the [doc^{FO30}](#) function. However, unlike the [doc^{FO30}](#) function, the [xsl:source-document](#) instruction offers no guarantee that the resulting document will be stable (that is, that multiple calls specifying the same URI will return the same document).

Specifically, if an [xsl:source-document](#) instruction is evaluated several times (or if different [xsl:source-document](#) instructions are evaluated) with the same URI (after making it absolute) as the value of the `href` attribute, it is [implementation-dependent](#) whether the same nodes or different nodes are returned on each occasion; it is also possible that the actual document content will be different.

Note:

A different node will necessarily be returned if there are differences in attributes such as `validation`, `type`, `streamable`, or `use-accumulators`, or if the calls are in different `packages` with variations in the rules for whitespace stripping or stripping of type annotations.

The result of the `xsl:source-document` instruction is the same as the result of the following (non-streaming) process:

1. The source document is read from the supplied URI and parsed to form a tree of nodes in the XDM data model.
2. The contained sequence constructor is evaluated with the root node of this tree as the context item, and with the context position and context size set to one; and the resulting sequence is returned as the result of the `xsl:source-document` instruction.

The `xsl:source-document` instruction is guaranteed-streamable if both the following conditions are satisfied:

1. It is declared-streamable, by specifying `streamable="yes"`.
2. the contained sequence constructor is grounded, as assessed using the streamability analysis in [19 Streamability](#). The consequences of being or not being guaranteed streamable depend on the processor conformance level, and are explained in [19.10 Streamability Guarantees](#).

The `use-accumulators` attribute defines the set of accumulators that are applicable to the document, as explained in [18.2.2 Applicability of Accumulators](#).

Note:

The following notes apply specifically to streamed processing.

The rules for guaranteed streamability ensure that the sequence constructor (and therefore the `xsl:source-document` instruction) cannot return any nodes from the streamed document. For example, it cannot contain the instruction `<xsl:sequence select="//chapter"/>`. If nodes from this document are to be returned, they must first be copied, for example by using the `xsl:copy-of` instruction or by calling the `copy-of` or `snapshot` functions.

Because the `xsl:source-document` instruction cannot (if it satisfies the rules for guaranteed streamability) return nodes from the streamed document, any nodes it does return will be conventional (unstreamed) nodes that can be processed without restriction. For example, if `xsl:source-document` is invoked within a stylesheet function `f:firstChapter`, and the sequence constructor consists of the instruction `<xsl:copy-of select="//chapter"/>`, then the calling code can manipulate the resulting `chapter` elements as ordinary trees rooted at parentless element nodes.

If the sequence constructor in an `xsl:source-document` instruction were to return nodes from the document for which streaming has been requested, the instruction would not be guaranteed streamable. Processors which support the streaming feature would then not be required to process it in a streaming manner, and this specification imposes no restrictions on the processing of the nodes returned. (The ability of a streaming processor to handle such stylesheets in a streaming manner might, of course, depend on how the nodes returned are processed, but those details are out of scope for this specification.)

18.1.1 Validation of Source Documents

The **validation** and **type** attributes of [`xsl:source-document`](#) may be used to control schema validation of the input document. They have the same effect as the corresponding attributes of the [`xsl:copy-of`](#) instruction when applied to a document node, except that when `streamable="yes"` is specified, the copy that is produced is itself a streamed document. The process is described in more detail in [25.4.2 Validating Document Nodes](#).

These two attributes are both optional, and if one is specified then the other **MUST** be omitted ([see [ERR XTSE1505](#)]).

The presence of a **validation** or **type** attribute on an [`xsl:source-document`](#) instruction causes any **input-type-annotations** attribute to have no effect on any document read using that instruction.

Note:

In effect, setting **validation** to **strict** or **lax**, or supplying the **type** attribute, requests document-level validation of the input as it is read. Setting **validation="preserve"** indicates that if the incoming document contains type annotations (for example, produced by validating the output of a previous step in a streaming pipeline) then they should be retained, while the value **strip** indicates that any such type annotations should be dropped.

It is a consequence of the way validation is defined in XSD that the type annotation of an element node can be determined during the processing of its start tag, although the actual validity of the element is not known until the end tag is encountered. When validation is requested, a streamed document should not present data to the stylesheet except to the extent that such data could form the leading part of a valid document. If the document proves to be invalid, the processor should not pass invalid data to the stylesheet to be processed, but should immediately signal the appropriate error. For the purposes of [`xsl:try`](#) and [`xsl:catch`](#), this error can only be caught at the level of the [`xsl:source-document`](#) instruction that initiated validation, not at a finer level. If validation errors are caught in this way, any output that has been computed up to the point of the error is not added to the final result tree; the mechanisms to achieve this may use memory, which may reduce the efficacy of streaming.

The analysis of guaranteed streamability (see [19 Streamability](#)) takes no account of information that might be obtained from a schema-aware static analysis of the stylesheet. Implementations may, however, be able to use streaming strategies for stylesheets that are not guaranteed-streamable, by taking advantage of such information. For example, an implementation might be able to treat the expression `.//title` as [striding](#) rather than [crawling](#) if it can establish from knowledge of the schema that two `title` elements will never be nested one inside the other.

18.1.2 Examples of `xsl:source-document`

The [`xsl:source-document`](#) instruction can be used to initiate processing of a document using streaming with a variety of coding styles, illustrated in the examples below.

Example: Using `xsl:source-document` with Aggregate Functions

The following example computes the number of transactions in a transaction file

Input:

```
<transactions>
  <transaction value="12.51"/>
  <transaction value="3.99"/>
</transactions>
```

Stylesheet code:

```
<xsl:source-document streamable="yes" href="transactions.xml">
  <count>
    <xsl:value-of select="count(transactions/transaction)"/>
  </count>
</xsl:source-document>
```

Result:

```
<count>2</count>
```

Analysis:

1. The literal result element `count` has the same sweep as the `xsl:value-of` instruction.
2. The `xsl:value-of` instruction has the same sweep as its `select` expression.
3. The call to `count` has the same sweep as its argument.
4. The argument to `count` is a `RelativePathExpr`. Under the rules in [19.8.8.8 Streamability of Path Expressions](#), this expression is `striding` and `consuming`. The call on `count` is therefore `grounded` and `consuming`.
5. The entire body of the `xsl:source-document` instruction is therefore `grounded` and `consuming`.

The following example computes the highest-value transaction in the same input file:

```
<xsl:source-document streamable="yes" href="transactions.xml">
  <maxValue>
    <xsl:value-of select="max(transactions/transaction/@value)"/>
  </maxValue>
</xsl:source-document>
```

Result:

```
<maxValue>12.51</maxValue>
```

Analysis:

1. The literal result element `maxValue` has the same sweep as the `xsl:value-of` instruction.
2. The `xsl:value-of` instruction has the same sweep as its `select` expression.
3. The call to `max` has the same sweep as its argument.

4. The argument to `max` is a `RelativePathExpr` whose two operands are the `RelativePathExpr` `transactions/transaction` and the `AxisStep @value`. The left-hand operand `transactions/transaction` has striding posture. The right-hand operand `@value`, given that it appears in a node value context, is motionless. The `RelativePathExpr` argument to `max` is therefore consuming.
5. The entire body of the `xsl:source-document` instruction is therefore consuming.

To compute both the count and the maximum value in a single pass over the input, several approaches are possible. The simplest is to use maps (map constructors are exempt from the usual rule that multiple downward selections are not allowed):

```
<xsl:source-document streamable="yes" href="transactions.xml">
  <xsl:variable name="tally" select="map{ 'count':
    count(transactions/transaction),
    'max':
    max(transactions/transaction/@value)}"/>
  <value count="${tally('count')}" max="${tally('max')}"/>
</xsl:source-document>
```

Other options include the use of `xsl:fork`, or multiple `xsl:accumulator` declarations, one for each value to be computed.

Example: Using `xsl:source-document` with `xsl:for-each` to Process a Collection of Input Documents

This example displays a list of the chapter titles extracted from each book in a collection of books.

Each input document is assumed to have a structure such as:

```
<book>
  <chapter number-of-pages="18">
    <title>The first chapter of book A</title>
    ...
  </chapter>
  <chapter number-of-pages="15">
    <title>The second chapter of book A</title>
    ...
  </chapter>
  <chapter number-of-pages="12">
    <title>The third chapter of book A</title>
    ...
  </chapter>
</book>
```

Stylesheet code:

```
<chapter-titles>
  <xsl:for-each select="uri-collection('books')">
    <xsl:source-document streamable="yes" href=". ">
      <xsl:for-each select="book">
        <xsl:for-each select="chapter">
          <title><xsl:value-of select="title"/></title>
        </xsl:for-each>
      </xsl:for-each>
    </xsl:source-document>
  </xsl:for-each>
</chapter-titles>
```

Output:

```
<chapter-titles>
  <title>The first chapter of book A</title>
  <title>The second chapter of book A</title>
  ...
  <title>The first chapter of book B</title>
  ...
</chapter-titles>
```

Note:

This example uses the function `uri-collection`^{F030} to obtain the document URIs of all the documents in a collection, so that each one can be processed in turn using `xsl:source-document`.

Example: Using `xsl:source-document` with `xsl:iterate`

This example assumes that the input is a book with multiple chapters, as shown in the previous example, with the page count for each chapter given as an attribute of the chapter. The transformation determines the starting page number for each chapter by accumulating the page counts for previous chapters, and rounding up to an odd number if necessary.

```
<chapter-start-page>
  <xsl:source-document streamable="yes" href="book.xml">
    <xsl:iterate select="book/chapter">
      <xsl:param name="start-page" select="1"/>
      <chapter title="{title}" start-page="{{$start-page}}"/>
      <xsl:next-iteration>
        <xsl:with-param name="start-page"
          select="$start-page + @number-of-pages +
          (@number-of-pages mod 2)"/>
      </xsl:next-iteration>
    </xsl:iterate>
  </xsl:source-document>
</chapter-start-page>
```

Output:

```
<chapter-start-page>
  <chapter title="The first chapter of book A" start-page="1"/>
  <chapter title="The second chapter of book A" start-page="19"/>
  <chapter title="The third chapter of book A" start-page="35"/>
  ...
</chapter-start-page>
```

Example: Using `xsl:source-document` with `xsl:for-each-group`

This example assumes that the input is a book with multiple chapters, and that each chapter belongs to a part, which is present as an attribute of the chapter (for example, chapters 1-4 might constitute Part 1, the next three chapters forming Part 2, and so on):

```
<book>
  <chapter part="1">
    <title>The first chapter of book A</title>
    ...
  </chapter>
  <chapter part="1">
    <title>The second chapter of book A</title>
    ...
  </chapter>
  ...
  <chapter part="2">
    <title>The fifth chapter of book A</title>
    ...
  </chapter>
</book>
```

The transformation copies the full text of the chapters, creating an extra level of hierarchy for the parts.

```
<book>
  <xsl:source-document streamable="yes" href="book.xml">
    <xsl:for-each select="book">
      <xsl:for-each-group select="chapter" group-adjacent="data(@part)">
        <part number="{current-grouping-key()}">
          <xsl:copy-of select="current-group()"/>
        </part>
      </xsl:for-each-group>
    </xsl:for-each>
  </xsl:source-document>
</book>
```

Output:

```
<book>
  <part number="1">
    <chapter part="1">
      <title>The first chapter of book A</title>
      ...
    </chapter>
    <chapter part="1">
      <title>The second chapter of book A</title>
      ...
    </chapter>
    ...
  </part>
  <part number="2">
    <chapter part="2">
      <title>The fifth chapter of book A</title>
      ...
    </chapter>
    ...
  </part>
</book>
```

Example: Using [xsl:source-document](#) with [xsl:apply-templates](#)

This example copies an XML document while deleting all the `ednote` elements at any level of the tree, together with their descendants. This example is a complete stylesheet, which is intended to be evaluated by nominating `main` as the [initial named template](#). The use of `on-no-match="deep-copy"` in the [xsl:mode](#) declaration means that the built-in template rule copies nodes unchanged, except where overridden by a user-defined template rule.

```
<xsl:transform version="3.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:mode name="delete-ednotes" streamable="yes"
    on-no-match="shallow-copy"/>

<xsl:template name="main">
    <xsl:source-document streamable="yes" href="book.xml">
        <xsl:apply-templates mode="delete-ednotes"/>
    </xsl:source-document>
</xsl:template>

<xsl:template match="ednote" mode="delete-ednotes"/>

</xsl:transform>
```

Additional template rules could be added to process other elements and attributes in the same pass through the data: for example, to modify the value of a `last-updated` attribute (wherever it appears) to the current date and time, the following rule suffices:

```
<xsl:template match="@last-updated" mode="delete-ednotes">
    <xsl:attribute name="last-updated" select="current-dateTime()"/>
</xsl:template>
```

18.1.3 [fn:stream-available](#)

Summary

Determines, as far as possible, whether a document is available for streamed processing using [xsl:source-document](#).

Signature

```
fn:stream-available($uri as xs:string?) as xs:boolean
```

Properties

This function is [nondeterministic](#)^{FO30}, [context-dependent](#)^{FO30}, and [focus-independent](#)^{FO30}. It depends on available documents.

Rules

The intent of the [stream-available](#) function is to allow a stylesheet author to determine, before calling [xsl:source-document](#) with `streamable="yes"` and with a particular URI as the value of its `href` attribute, whether a document is available at that location for streamed processing.

If the `$uri` argument is an empty sequence then the function returns `false`.

If the function returns `true` then the caller can conclude that the following conditions are true:

1. The supplied URI is valid;
2. A resource can be retrieved at that URI;
3. An XML representation of the resource can be delivered, which is well-formed at least to the extent that some initial sequence of octets can be decoded into characters and matched against the production:
`prolog (EmptyElemTag | STag)`
as defined in the XML 1.0 or XML 1.1 Recommendation.

Note:

That is, the XML is well-formed at least as far as the end of the first element start tag; to establish this, a parser will typically retrieve any external entities referenced in the Doctype declaration or DTD.

If the function returns `false`, the caller can conclude that either one of the above conditions is not satisfied, or the processor detected some other condition that would prevent a call on [`xsl:source-document`](#) with `streamable="yes"` executing successfully.

Like [`xsl:source-document`](#) itself, the function is not deterministic, which means that multiple calls during the execution of a stylesheet will not necessarily return the same result. The caller cannot make any inferences about the point in time at which the input conditions for [`stream-available`](#) are present, and in particular there is no guarantee that because [`stream-available`](#) returns true, [`xsl:source-document`](#) will necessarily succeed.

The value of the `$uri` argument **MUST** be a URI in the form of a string. If it is a relative URI, it is resolved relative to the static base URI of the function call.

Error Conditions

If the URI is invalid, such that a call on [`doc-available`](#)^{FO30} would signal an error, then [`stream-available`](#) signals the same error: [`\[ERR FODC0005\]`](#)^{FO30}.

18.2 Accumulators

Accumulators are introduced in XSLT 3.0 to enable data that is read during streamed processing of a document to be accumulated, processed or retained for later use. However, they may equally be used with non-streamed processing.

[**DEFINITION:** An **accumulator** defines a series of values associated with the nodes of the tree. If an accumulator is applicable to a particular tree, then for each node in the tree, other than attribute and namespace nodes, there will be two values available, called the pre-descent and post-descent values. These two values are available via a pair of functions, [`accumulator-before`](#) and [`accumulator-after`](#).]

There are two ways the values of an accumulator can be established for a given tree: they can be computed by evaluating the rules appearing in the [`xsl:accumulator`](#) declaration, or they can be copied from the corresponding nodes in a different tree. The second approach (copying the values) is available via the [`snapshot`](#) and [`copy-of`](#) functions, or by use of the [`xsl:copy-of`](#) instruction specifying `copy-accumulators="yes"`. Accumulator

values are also copied during the implicit invocation of the snapshot function performed by the [xsl:merge](#) instruction.

Note:

Accumulators can apply to trees rooted at any kind of node. But because they are most often applied to trees rooted at a document node, this section sometimes refers to the “document” to which an accumulator applies; use of this term should be taken to include all trees whether or not they are rooted at a document node.

Accumulators can apply to trees rooted at nodes (such as text nodes) that cannot have children, though this serves no useful purpose. In the case of a tree rooted at an attribute or namespace node, there is no way to obtain the value of the accumulator.

The following sections give first, the syntax rules for defining an accumulator; then an informal description of the semantics; then a more formal definition; and finally, examples. But to illustrate the concept intuitively, the following simple example shows how an accumulator can be used for numbering of nodes:

Example: Numbering Figures within a Chapter

This example assumes document input in which `figure` elements can appear within `chapter` elements (which we assume are not nested), and the requirement is to render the figures with a caption that includes the figure number within its containing chapter.

When the document is processed using streaming, the [xsl:number](#) instruction is not available, so a solution using accumulators is needed.

The required accumulator can be defined and used like this:

```
<xsl:accumulator name="figNr" as="xs:integer"
                  initial-value="0" streamable="yes">
  <xsl:accumulator-rule match="chapter" select="0"/>
  <xsl:accumulator-rule match="figure" select="$value + 1"/>
</xsl:accumulator>

<xsl:mode streamable="yes"/>
<xsl:template match="figure">
  <xsl:apply-templates/>
  <p>Figure <xsl:value-of select="accumulator-before('figNr')"/></p>
</xsl:template>
```

18.2.1 Declaring an Accumulator

```
<!-- Category: declaration -->
<xsl:accumulator
  name = eqname
  initial-value = expression
  as? = sequence-type
  streamable? = boolean >
  <!-- Content: xsl:accumulator-rule+ -->
</xsl:accumulator>
```

```
<xsl:accumulator-rule
  match = pattern
  phase? = "start" | "end"
  select? = expression >
  <!-- Content: sequence-constructor -->
</xsl:accumulator-rule>
```

An [xsl:accumulator](#) element is a [declaration](#) of an accumulator. The `name` attribute defines the name of the accumulator. The value of the `name` attribute is an [EQName](#), which is expanded as described in [5.1.1 Qualified Names](#).

An [xsl:accumulator](#) declaration can only appear as a [top-level](#) element in a stylesheet module.

The functions [accumulator-before](#) and [accumulator-after](#) return, respectively, the value of the accumulator before visiting the descendants of a given node, and the value after visiting the descendants of a node. Each of these functions takes a single argument, the name of the accumulator, and the function applies implicitly to the context node. The type of the return value (for both functions) is determined by the `as` attribute of the [xsl:accumulator](#) element.

[**DEFINITION:** The functions [accumulator-before](#) and [accumulator-after](#) are referred to as the **accumulator functions**.]

For constructs that use accumulators to be [guaranteed-streamable](#):

- The [accumulator-before](#) function for a streamed node can be called at any time the node is available (it behaves like other properties of the node such as `name`^{FO30} or `base-uri`^{FO30}).
- The [accumulator-after](#) function, however, is restricted to appear after any instruction that reads the descendants of the node in question. The constraints are expressed as static rules: see [19.8.9.1 Streamability of the accumulator-after Function](#) for more details.

The initial value of the accumulator is obtained by evaluating the expression in the `initial-value` attribute. This attribute is mandatory. The expression in the `initial-value` attribute is evaluated with a [singleton focus](#) based on the root node of the streamed input tree to which the accumulator is being applied.

The values of the accumulator for individual nodes in a tree are obtained by applying the [xsl:accumulator-rule](#) rules contained within the [xsl:accumulator](#) declaration, as described in subsequent sections. The `match` attribute of [xsl:accumulator-rule](#) is a [pattern](#) which determines which nodes trigger execution of the rule; the `phase` attribute indicates whether the rule fires before descendants are processed (`phase="start"`, which is the default), or after descendants are processed (`phase="end"`).

The `select` attribute and the contained sequence constructor of the `xsl:accumulator-rule` element are mutually exclusive: if the `select` attribute is present then the sequence constructor must be empty. The expression in the `select` attribute of `xsl:accumulator-rule` or the contained sequence constructor is evaluated with a static context that follows the normal rules for expressions in stylesheets, except that:

- An additional variable is present in the context. The name of this variable is `value` (in no namespace), and its type is the type that appears in the `as` attribute of the `xsl:accumulator` declaration.
- The context item for evaluation of the expression or sequence constructor will always be a node that matches the `pattern` in the `match` attribute.

The result of both the `initial-value` and `select` expressions (or contained sequence constructor) is converted to the type declared in the `as` attribute by applying the [function conversion rules](#). A [type error](#) occurs if conversion is not possible. The `as` attribute defaults to `item()*`.

The effect of the `streamable` attribute is defined in [18.2.9 Streamability of Accumulators](#).

[18.2.2 Applicability of Accumulators](#)

It is not the case that every accumulator is applicable to every tree. The details depend on how the accumulator is declared, and how the tree is created. The rules are as follows:

1. An accumulator is applicable to a tree unless otherwise specified in these rules. (For example, when a document is read using the `document`, `doc`^{FO30}, or `collection`^{FO30} functions, all accumulators are applicable. Similarly, all accumulators are applicable to a `temporary tree` created using `xsl:variable`.)
2. Regardless of the rules below, an accumulator is not applicable to a `streamed document` unless the accumulator is declared with `streamable="yes"`. (The converse does not apply: for unstreamed documents, accumulators are applicable regardless of the value of the `streamable` attribute.)
3. For a document read using the `xsl:source-document` instruction, the accumulators that are applicable are those determined by the `use-accumulators` attribute of that instruction.
4. For a document read using the `for-each-source` attribute of an `xsl:merge-source` child of an `xsl:merge` instruction, the accumulators that are applicable are those determined by the `use-accumulators` attribute of the `xsl:merge-source` element.
5. For a document containing nodes supplied in the `initial match selection`, the accumulators that are applicable are those determined by the `xsl:mode` declaration of the `initial mode`. This means that in the absence of an `xsl:mode` declaration, no accumulators are applicable.
6. For a tree T created by copying a node in a tree S using the `copy-of` or `snapshot` functions, or the instruction `xsl:copy-of` with `copy-accumulators="yes"`, an accumulator is applicable to T if and only if it is applicable to S .

If an accumulator is not applicable to the tree containing the context item, calls to the functions `accumulator-before` and `accumulator-after`, supplying the name of that accumulator, will fail with a dynamic error.

Note:

The reason that accumulators are not automatically applicable to every streamed document is to avoid the cost of evaluating them, and to avoid the possibility of dynamic errors occurring if they are not designed to work with a particular document structure.

In the case of unstreamed documents, there are no compelling reasons to restrict which accumulators are applicable, because an implementation can avoid the cost of evaluating every accumulator against every document by evaluating the accumulator lazily, for example, by only evaluating the accumulator for a particular tree the first time its value is requested for a node in that tree. In the interests of orthogonality, however, restricting the applicable accumulators works in the same way for streamable and non-streamable documents.

The value of the `use-accumulators` attribute of [xsl:source-document](#), [xsl:merge-source](#), or [xsl:mode](#) must either a whitespace-separated list of [EQNames](#), or the special token `#all`. The list may be empty, and the default value is an empty list. Every EQName in the list must be the name of an accumulator, visible in the containing package, and declared with `streamable="yes"`. The value `#all` indicates that all accumulators that are visible in the containing package are applicable (except that for a streamable input document, an accumulator is not applicable unless it specifies `streamable="yes"`).

[ERR XTSE3300] It is a [static error](#) if the list of accumulator names contains an invalid token, contains the same token more than once, or contains the token `#all` along with any other value; or if any token (other than `#all`) is not the name of a [declared-streamable](#) accumulator visible in the containing package.

18.2.3 Informal Model for Accumulators

This section describes how accumulator values are established by evaluating the rules in an [xsl:accumulator](#) declaration. This process does not apply to trees created with accumulator values copied from another document, for example by using the [copy-of](#) or [snapshot](#) functions.

Informally, an accumulator is evaluated by traversing a tree, as follows.

Each node is visited twice, once before processing its descendants, and once after processing its descendants. For consistency, this applies even to leaf nodes: each is visited twice. Attribute and namespace nodes, however, are not visited.

Before the traversal starts, a variable (called the accumulator variable) is initialized to the value of the expression given as the `initial-value` attribute.

On each node visit, the [xsl:accumulator-rule](#) elements are examined to see if there is a matching rule. For a match to occur, the pattern in the `match` attribute must match the node, and the `phase` attribute must be `start` if this is the first visit, and `end` if it is the second visit. If there is a matching rule, then a new value is computed for the accumulator variable using the expression contained in that rule's `select` attribute or the contained sequence constructor. If there is more than one matching rule, the last in document order is used. If there is no matching rule, the value of the accumulator variable does not change.

Each node is labeled with a pre-descent value for the accumulator, which is the value of the accumulator variable immediately *after* processing the first visit to that node, and with a post-descent value for the accumulator, which is the value of the accumulator variable immediately *after* processing the second visit.

The function `accumulator-before` delivers the pre-descent value of the accumulator at the context node; the function `accumulator-after` delivers the post-descent value of the accumulator at the context node.

Although this description is expressed in procedural terms, it can be seen that the two values of the accumulator for any given node depend only on the node and its preceding and (in the case of the post-descent value) descendant nodes. Calculation of both values is therefore deterministic and free of side-effects; moreover, it is clear that the values can be computed during a streaming pass of a document, provided that the rules themselves use only information that is available without repositioning the input stream.

It is permitted for the `select` expression of an accumulator rule, or the contained sequence constructor, to invoke an accumulator function. For a streamable accumulator, the rules ensure that a rule with `phase="start"` cannot call the `accumulator-after` function. When such function calls exist in an accumulator rule, they impose a dependency of one accumulator on another, and create the possibility of cyclic dependencies. Processors are allowed to report the error statically if they can detect it statically. Failing this, processors are allowed to fail catastrophically in the event of a cycle, in the same way as they might fail in the event of infinite function or template recursion. Catastrophic failure might manifest itself, for example, as a stack overflow, or as non-termination of the transformation.

18.2.4 Formal Model for Accumulators

This section describes how accumulator values are established by evaluating the rules in an `xsl:accumulator` declaration. This process does not apply to trees created with accumulator values copied from another document, for example by using the `copy-of` or `snapshot` functions.

[**DEFINITION:** A **traversal** of a tree is a sequence of `traversal events`.]

[**DEFINITION:** a **traversal event** (shortened to **event** in this section) is a pair comprising a phase (start or end) and a node.] It is modelled as a map with two entries: `map{"phase": p, "node": n}` where p is the string "start" or "end" and n is a node.

The traversal of a tree contains two traversal events for each node in the tree, other than attribute and namespace nodes. One of these events (the "start event") has phase = "start", the other (the "end event") has phase = "end".

The order of traversal events within a traversal is such that, given any two nodes M and N with start/end events denoted by M_0, M_1, N_0 , and N_1 , :

- For any node N, N_0 precedes N_1 ;
- If M is an ancestor of N then M_0 precedes N_0 and N_1 precedes M_1 ;
- If M is on the preceding axis of N then M_1 precedes N_0 .

The accumulator defines a (private) delta function Δ . The delta function computes the value of the accumulator for one traversal event in terms of its value for the previous traversal event. The function is defined as follows:

1. The signature of Δ is `function ($old-value as T, $event as map(*)) as T`, where T is the sequence type declared in the `as` attribute of the accumulator declaration;
2. The implementation of the function is equivalent to the following algorithm:
 - a. Let R be the set of `xsl:accumulator-rule` elements among the children of the accumulator declaration whose `phase` attribute equals `$event("phase")` and whose `match` attribute is a `pattern` that matches `$event("node")`

- b. If R is empty, return $\$old\text{-}value$
- c. Let Q be the [`xsl:accumulator-rule`](#) in R that is last in document order
- d. Return the value of the expression in the `select` attribute of Q , or the contained sequence constructor, evaluating this with a [`singleton focus`](#) set to $\$event("node")$ and with a dynamic context that binds the variable whose name is $\$value$ (in no namespace) to the value $\$old\text{-}value$.

Note:

The argument names `old-value` and `event` are used here purely for definitional purposes; these names are not available for use within the `select` expression or contained sequence constructor.

For every node N , other than attribute and namespace nodes, the accumulator defines a pre-descent value B_N and a post-descent value A_N whose values are as follows:

1. Let T be the [`traversal`](#) of the tree rooted at $\text{fn:root}(N)$.
2. Let SB be the subsequence of T starting at the first event in T and ending with the start event for node N (that is, the event `map{ "phase": "start", "node": N }`).
3. Let SA be the subsequence of T starting at the first event in T , and ending with the end event for node N (that is, the event `map{ "phase": "end", "node": N }`).
4. Let Z be the result of evaluating the expression contained in the `initial-value` attribute of the [`xsl:accumulator`](#) declaration, evaluated with a [`singleton focus`](#) based on $\text{root}(N)$.
5. Then the pre-descent value B_N is the value of $\text{fn:fold-left}(SB, Z, \Delta)$, and the post-descent value A_N is the value of $\text{fn:fold-left}(SA, Z, \Delta)$.

18.2.5 [Dynamic Errors in Accumulators](#)

If a dynamic error occurs when evaluating the `initial-value` expression of [`xsl:accumulator`](#), or the `select` expression of [`xsl:accumulator-rule`](#), then the error is signaled as an error from any subsequent call on [`accumulator-before`](#) or [`accumulator-after`](#) that references the accumulator. If no such call on [`accumulator-before`](#) or [`accumulator-after`](#) happens, then the error goes unreported.

Note:

In the above rule, the phrase **subsequent call** is to be understood in terms of functional dependency; that is, a call to [`accumulator-before`](#) or [`accumulator-after`](#) signals an error if the accumulator value at the node in question is functionally dependent on a computation that fails with a dynamic error.

Note:

Particularly in the case of streamed accumulators, this may mean that the implementation has to “hold back” the error until the next time the accumulator is referenced, to give applications the opportunity to catch the error using [`xsl:try`](#) and [`xsl:catch`](#) in a predictable way.

Note:

Errors that occur during the evaluation of the pattern in the `match` attribute of [xsl:accumulator-rule](#) are handled as described in [5.5.4 Errors in Patterns](#): specifically, the pattern does not match the relevant node, and no error is signaled.

18.2.6 [fn:accumulator-before](#)

Summary

Returns the pre-descent value of the selected accumulator at the context node.

Signature

```
fn:accumulator-before($name as xs:string) as item()*
```

Properties

This function is [deterministic^{FO30}](#), [context-dependent^{FO30}](#), and [focus-dependent^{FO30}](#).

Rules

The `$name` argument specifies the name of the [accumulator](#). The value of the argument **MUST** be a string containing an [EQName](#). If it is a [lexical QName](#), then it is expanded as described in [5.1.1 Qualified Names](#) (no prefix means no namespace).

The function returns the pre-descent value $B(N)$ of the selected accumulator where N is the context node, as defined in [18.2.4 Formal Model for Accumulators](#).

If the context item is a node in a streamed document, then the accumulator must be declared with `streamable="yes"`.

Note:

The converse is not true: an accumulator declared to be streamable is available on both streamed and unstreamed nodes.

Error Conditions

[ERR XTDE3340] It is a [dynamic error](#) if the value of the first argument to the [accumulator-before](#) or [accumulator-after](#) function is not a valid [EQName](#), 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:accumulator](#) declaration appearing in the [package](#) in which the function call appears. 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 XTDE3350] It is a [dynamic error](#) to call the [accumulator-before](#) or [accumulator-after](#) function when there is no [context item](#).

[ERR XTTE3360] It is a [type error](#) to call the [accumulator-before](#) or [accumulator-after](#) function when the [context item](#) is not a node, or when it is an attribute or namespace node.

[ERR XTDE3362] It is a [dynamic error](#) to call the [accumulator-before](#) or [accumulator-after](#) function when the context item is a node in a tree to which the selected accumulator is not applicable (including the case where it is not applicable because the document is streamed and the accumulator is not declared with `streamable="yes"`). Implementations `MAY` raise this error but are `NOT REQUIRED` to do so, if they are capable of streaming documents without imposing this restriction.

[ERR XTDE3400] It is an error if there is a cyclic set of dependencies among accumulators such that the (pre- or post-descent) value of an accumulator depends directly or indirectly on itself. A processor `MAY` report this as a [static error](#) if it can be detected statically. Alternatively a processor `MAY` report this as a [dynamic error](#). As a further option, a processor may fail catastrophically when this error occurs.

Note:

The term **catastrophic failure** here means a failure similar to infinite function or template recursion, which might result in stack overflow or even in non-termination of the transformation, rather than in a dynamic error of the kind that can be processed using [`xsl:try`](#) and [`xsl:catch`](#).

Notes

The [accumulator-before](#) function can be applied to a node whether or not the accumulator has a `phase="start"` rule for that node. In effect, there is a `phase="start"` rule for every node, where the default rule is to leave the accumulator value unchanged; the [accumulator-before](#) function delivers the value of the accumulator after processing the explicit or implicit `phase="start"` rule.

Examples

Given the accumulator:

```
<xsl:accumulator name="a" initial-value="0">
  <xsl:accumulator-rule match="section" select="$value + 1"/>
</xsl:accumulator>
```

and the template rule:

```
<xsl:template match="section">
  <xsl:value-of select="accumulator-before('a')"/>
  <xsl:apply-templates/>
</xsl:template>
```

The stylesheet will precede the output from processing each section with a section number that runs sequentially 1, 2, 3... irrespective of the nesting of sections.

18.2.7 [fn:accumulator-after](#)

Summary

Returns the post-descent value of the selected accumulator at the context node.

Signature

```
fn:accumulator-after($name as xs:string) as item()*
```

Properties

This function is [deterministic^{FO30}](#), [context-dependent^{FO30}](#), and [focus-dependent^{FO30}](#).

Rules

The \$name argument specifies the name of the [accumulator](#). The value of the argument **MUST** be a string containing an [EQName](#). If it is a [lexical QName](#), then it is expanded as described in [5.1.1 Qualified Names](#) (no prefix means no namespace).

The function returns the post-descent value $A(N)$ of the selected accumulator where N is the context node, as defined in [18.2.4 Formal Model for Accumulators](#).

If the context item is a node in a streamed document, then the accumulator must be declared with `streamable="yes"`.

Note:

The converse is not true: an accumulator declared to be streamable is available on both streamed and unstreamed nodes.

Error Conditions

The following errors apply: [see [ERR_XTDE3340](#)], [see [ERR_XTDE3350](#)], [see [ERR_XTTE3360](#)], [see [ERR_XTDE3362](#)], [see [ERR_XTDE3400](#)].

For constraints on the use of [accumulator-after](#) when streaming, see [19.8.9.1 Streamability of the accumulator-after Function](#).

Notes

The [accumulator-after](#) function can be applied to a node whether or not the accumulator has a `phase="end"` rule for that node. In effect, there is a `phase="end"` rule for every node, where the default rule is to leave the accumulator value unchanged; the [accumulator-after](#) function delivers the value of the accumulator after processing the explicit or implicit `phase="end"` rule.

Examples

Given the accumulator:

```
<xsl:accumulator name="w" initial-value="0" streamable="true"
  as="xs:integer">
  <xsl:accumulator-rule match="text()"
    select="$value + count(tokenize(.))"/>
</xsl:accumulator>
```

and the template rule:

```
<xsl:template match="section">
  <xsl:apply-templates/>
  (words: <xsl:value-of select="accumulator-after('w') - accumulator-
  before('w')"/>)
</xsl:template>
```

The stylesheet will output at the end of each section a (crude) count of the number of words in that section.

Note: the call on tokenize(.) relies on XPath 3.1

18.2.8 Importing of Accumulators

If a [package](#) contains more than one [`xsl:accumulator`](#) declaration with a particular name, then the one with the highest [import precedence](#) is used.

[ERR XTSE3350] It is a [static error](#) for a [package](#) to contain two or more non-hidden accumulators with the same [expanded QName](#) and the same [import precedence](#), unless there is another accumulator with the same [expanded QName](#), and a higher import precedence.

Accumulators cannot be referenced from, or overridden in, a different package from the one in which they are declared.

18.2.9 Streamability of Accumulators

An accumulator is [guaranteed-streamable](#) if it satisfies all the following conditions:

1. The [`xsl:accumulator`](#) declaration has the attribute `streamable="yes"`.
2. In every contained [`xsl:accumulator-rule`](#), the [pattern](#) in the `match` attribute is a [motionless](#) pattern (see [19.8.10 Classifying Patterns](#)).
3. The [expression](#) in the `initial-value` attribute is [grounded](#) and [motionless](#).
4. The [expression](#) in the `select` attribute or contained sequence constructor is [grounded](#) and [motionless](#).

Specifying `streamable="yes"` on an [`xsl:accumulator`](#) element declares an intent that the accumulator should be streamable, either because it is [guaranteed-streamable](#), or because it takes advantage of streamability extensions offered by a particular processor. The consequences of declaring the accumulator to be streamable when it is not in fact guaranteed streamable depend on the conformance level of the processor, and are explained in [19.10 Streamability Guarantees](#).

When an accumulator is declared to be streamable, the stylesheet author must ensure that the accumulator function `accumulator-after` is only called at appropriate points in the processing, as explained in [19.8.9.1 Streamability of the accumulator-after Function](#).

18.2.10 Copying Accumulator Values

When nodes (including streamed nodes) are copied using the `snapshot` or `copy-of` functions, or using the `xsl:copy-of` instruction with the attribute `copy-accumulators="yes"`, then the pre-descent and post-descent values of accumulators for that tree are not determined by traversing the tree as described in [18.2.3 Informal Model for Accumulators](#) and [18.2.4 Formal Model for Accumulators](#). Instead the values are the same as the values on the corresponding nodes of the source tree.

This applies also to the implicit invocation of the `snapshot` function that happens during the evaluation of `xsl:merge`.

If an accumulator is not applicable to a tree S , then it is also not applicable to any tree formed by copying nodes from S using the above methods.

Note:

During streamed processing, accumulator values will typically be computed “on the fly”; when the `copy-of` or `snapshot` functions are applied to a streamed node, the computed accumulator values for the streamed document will typically be materialized and saved as part of the copy.

Accumulator values for a non-streamed document will often be computed lazily, that is, they will not be computed unless and until they are needed. A call on `copy-of` or `snapshot` on a non-streamed document whose accumulator values have not yet been computed can then be handled in a variety of ways. The implementation might interpret the call on `copy-of` or `snapshot` as a trigger causing the accumulator values to be computed; or it might retain a link between the nodes of the copied tree and the nodes of the original tree, so that a request for accumulator values on the copied tree can trigger computation of accumulator values for the original tree.

18.2.11 Examples of Accumulators

Example: Remember the Title of a Document

Consider an XHTML document in which the title of the document is represented by the content of a `title` element appearing as a child of the `head` element, which in turn appears as a child of the `html` element. Suppose that we want to process the document in streaming mode, and that we want to avoid outputting the content of the `h1` element if it is the same as the document title.

This can be achieved by remembering the value of the title in an accumulator variable.

```
<xsl:accumulator name="firstTitle" as="xs:string?" initial-value="()"  
    streamable="yes">  
    <xsl:accumulator-rule match="/html/head/title/text()" select="string(.)"/>  
</xsl:accumulator>
```

Subsequently, while processing an `h1` element appearing later in the document, the value can be referenced:

```
<xsl:template match="h1">  
    <xsl:variable name="firstTitle" select="accumulator-before('firstTitle')"/>  
    <xsl:variable name="thisTitle" select="string(.)"/>  
    <xsl:if test="$thisTitle ne $firstTitle">  
        <div class="heading-1"><xsl:value-of select="$thisTitle"/></div>  
    </xsl:if>  
</xsl:template>
```

Example: Keep a Word Count

Suppose that there is a requirement to output, at the end of the HTML rendition of a document, a paragraph giving the total number of words in the document.

An accumulator can be used to maintain a (crude) word count as follows:

```
<xsl:accumulator name="word-count"  
    as="xs:integer"  
    initial-value="0">  
    <xsl:accumulator-rule match="text()"  
        select="$value + count(tokenize(.))"/>  
</xsl:accumulator>
```

Note: the call on `tokenize` relies on XPath 3.1

The final value can be output at the end of the document:

```
<xsl:template match="/">  
    <xsl:apply-templates/>  
    <p>Word count: <xsl:value-of select="accumulator-after('word-count')"/>  
</p>  
</xsl:template>
```


Example: Output Hierarchic Section Numbers

Consider a document in which `section` elements are nested within `section` elements to arbitrary depth, and there is a requirement to render the document with hierarchic section numbers of the form 3.5.1.4.

The current section number can be maintained in an accumulator in the form of a sequence of integers, managed as a stack. The number of integers represents the current level of nesting, and the value of each integer represents the number of preceding sibling sections encountered at that level. For convenience the first item in the sequence represents the top of the stack.

```
<xsl:accumulator name="section-nr" as="xs:integer*"
    initial-value="0">
    <xsl:accumulator-rule match="section" phase="start"
        select="0, head($value)+1, tail($value)"/>
    <xsl:accumulator-rule match="section" phase="end"
        select="tail($value) (:pop:)"/>
</xsl:accumulator>
```

To illustrate this, consider the values after processing a series of start and end tags:

Example data illustrating the effect of parsing events on an accumulator

events	accumulator value	required section number
<section>	0, 1	1
<section>	0, 1, 1	1.1
</section>	1, 1	
<section>	0, 2, 1	1.2
</section>	2, 1	
<section>	0, 3, 1	1.3
<section>	0, 1, 3, 1	1.3.1
</section>	1, 3, 1	
<section>	0, 2, 3, 1	1.3.2
</section>	2, 3, 1	
</section>	3, 1	
</section>	1	

The section number for a section can thus be generated as:

```
<xsl:template match="section">
  <p>
    <xsl:value-of select="reverse(tail(accumulator-before('section-nr')))"
      separator=". "/>
  </p>
  <xsl:apply-templates/>
</xsl:template>
```

Example: Compute a Histogram showing the Number of Books, by Publisher

```
<xsl:accumulator name="histogram" as="map(xs:string, xs:integer)"
  initial-value="map{}">
  <xsl:accumulator-rule match="book">
    <xsl:choose>
      <xsl:when test="map:contains($value, @publisher)">
        <xsl:sequence select="map:put($value, string(@publisher),
          $value(@publisher)+1)"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:sequence select="map:put($value, string(@publisher), 1)"/>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:accumulator-rule>
</xsl:accumulator>
```

The contained `sequence` constructor is evaluated with the variable `$value` set to the current value, and with the context node as the node being visited.

Note:

In the two calls on `map:put()`, it is necessary to explicitly convert `@publisher` to an `xs:string` value, because this is the declared type of the keys in the result map. Relying on atomization would produce keys of type `xs:untypedAtomic`, which would not satisfy the declared type of the map.

The accumulated histogram might be displayed as follows:

```
<xsl:source-document streamable="yes" href="booklist.xml">
  ....
  <h1>Number of books, by publisher</h1>
  <table>
    <thead>
      <th>Publisher</th>
      <th>Number of books</th>
    </thead>
    <tbody>
      <xsl:variable name="histogram" select="accumulator-after('histogram')"/>
      <xsl:for-each select="map:keys($histogram)">
        <tr>
          <td><xsl:value-of select=". /></td>
          <td><xsl:value-of select="$histogram(.) /></td>
        </tr>
      </xsl:for-each>
    </tbody>
  </table>
</xsl:source-document>
```

18.3 fn:copy-of

Summary

Returns a deep copy of the sequence supplied as the `$input` argument, or of the context item if the argument is absent.

Signatures

```
fn:copy-of() as item()
```

```
fn:copy-of($input as item()* ) as item()*
```

Properties

The zero-argument form of this function is [nondeterministic^{FO30}](#), [focus-dependent^{FO30}](#), and [context-independent^{FO30}](#).

The one-argument form of this function is [nondeterministic^{FO30}](#), [focus-independent^{FO30}](#), and [context-independent^{FO30}](#).

Rules

The zero-argument form of this function is defined so that `copy-of()` returns the value of `internal:copy-item(.)`, where `internal:copy-item` (which exists only for the purpose of this exposition) is defined below. Informally, `copy-of()` copies the context item.

The single argument form of this function is defined in terms of the `internal:copy-item` as follows: `copy-of($input)` is equivalent to `$input ! internal:copy-item(.)`. Informally, `copy-of($input)` copies each item in the input sequence in turn.

The `internal:copy-item` function is defined as follows:

```
<xsl:function name="internal:copy-item" as="item()">
  new-each-time="maybe">
  <xsl:param name="input" as="item()"/>
  <xsl:copy-of select="$input"
    copy-namespaces="yes"
    copy-accumulators="yes"
    validation="preserve"/>
</xsl:function>
```

The streamability analysis, however, is different: see [19.8.9 Classifying Calls to Built-In Functions](#).

The use of `new-each-time="maybe"` in the above definition means that if the `internal:copy-item` function is called more than once with the same node as argument (whether or not these calls are part of the same call on `copy-of`), then it is [implementation-dependent](#) whether each call returns the same node, or whether multiple calls return different nodes. Returning the original node, however, is not allowed, except as an optimization when the processor can determine that this is equivalent.

Note:

One case where such optimization might be possible is when the copy is immediately atomized.

Notes

The [copy-of](#) function is available for use (and is primarily intended for use) when a source document is processed using streaming. It can also be used when not streaming. The effect, when applied to element and document nodes, is to take a copy of the subtree rooted at the current node, and to make this available as a normal tree: one that can be processed without any of the restrictions that apply while streaming, for example only being able to process children once. The copy, of course, does not include siblings or ancestors of the context node, so any attempt to navigate to siblings or ancestors will result in an empty sequence being returned.

All nodes in the result sequence will be parentless.

If atomic values or functions (including maps and arrays) are present in the input sequence, they will be included unchanged at the corresponding position of the result sequence.

Accumulator values are taken from the copied document as described in [18.2.10 Copying Accumulator Values](#).

Examples

Using `copy-of()` while streaming:

This example copies from the source document all employees who work in marketing and are based in Dubai. Because there are two accesses using the child axis, it is not possible to do this without buffering each employee in memory, which can be achieved using the [copy-of](#) function.

```
<xsl:source-document streamable="yes" href="employees.xml">
  <xsl:sequence select="copy-of(employees/employee)
    [department='Marketing' and location='Dubai']"/>
</xsl:source-document>
```

[18.4 fn:snapshot](#)

Summary

Returns a copy of a sequence, retaining copies of the ancestors and descendants of any node in the input sequence, together with their attributes and namespaces.

Signatures

`fn:snapshot() as item()`

`fn:snapshot($input as item()*) as item()*`

Properties

The zero-argument form of this function is [nondeterministic^{FO30}](#), [focus-dependent^{FO30}](#), and [context-independent^{FO30}](#).

The one-argument form of this function is nondeterministic^{FO30}, focus-independent^{FO30}, and context-independent^{FO30}.

Rules

The zero-argument form of this function is defined so that `snapshot()` returns the value of `internal:snapshot-item(.)`, where `internal:snapshot-item` (which exists only for the purpose of this exposition) is defined below. Informally, `snapshot()` takes a snapshot of the context item.

The single argument form of this function is defined in terms of the `internal:snapshot-item` as follows: `snapshot($input)` is equivalent to `$input ! internal:snapshot-item(.)`. Informally, `snapshot($input)` takes a snapshot of each item in the input sequence in turn.

The `internal:snapshot-item` function behaves as follows:

- If the supplied item is an atomic value or a function item (including maps and arrays), then it returns that item unchanged.
- If the supplied item is a node, then it returns a snapshot of that node, as defined below.

[**DEFINITION:** A **snapshot** of a node N is a deep copy of N , as produced by the `xsl:copy-of` instruction with `copy-namespaces` set to `yes`, `copy-accumulators` set to `yes`, and `validation` set to `preserve`, with the additional property that for every ancestor of N , the copy also has a corresponding ancestor whose name, `node-kind`, and base URI are the same as the corresponding ancestor of N , and that has copies of the attributes, namespaces and accumulator values of the corresponding ancestor of N . But the ancestor has a type annotation of `xs:anyType`, has the properties `nilled`, `is-id`, and `is-idref` set to `false`, and has no children other than the child that is a copy of N or one of its ancestors.]

If the function is called more than once with the same argument, it is implementation-dependent whether each call returns the same node, or whether multiple calls return different nodes. That is, the result of the expression `snapshot($X) is snapshot($X)` is implementation-dependent.

Except for the effect on accumulators, the `internal:snapshot-item` function can be expressed as follows:

```

<xsl:function name="internal:snapshot-item" as="item()">
    <xsl:param name="input" as="item()"/>
    <xsl:apply-templates select="$input" mode="internal:snapshot"/>
</xsl:function>

<!-- for atomic values and function items, return the item unchanged -->

<xsl:template match=". " mode="internal:snapshot" priority="1">
    <xsl:sequence select=". "/>
</xsl:template>

<!-- for a document node, or any other root node, return a deep copy -->

<xsl:template match="root()" mode="internal:snapshot" priority="5">
    <xsl:copy-of select=". "/>
</xsl:template>

<!-- for an element, comment, text node, or processing instruction: -->

<xsl:template match="node()" mode="internal:snapshot"
               as="node()" priority="3">
    <xsl:sequence select="internal:graft-to-parent(
        ., ..., function($n){$n/node()} )"/>
</xsl:template>

<!-- for an attribute: -->

<xsl:template match="@*" mode="internal:snapshot"
               as="attribute()" priority="3">
    <xsl:variable name="name" select="node-name(.)"/>
    <xsl:sequence select="internal:graft-to-parent(., ...
        function($n){$n/@*[node-name(.) = $name]} )"/>
</xsl:template>

<!-- for a namespace node: -->

<xsl:template match="namespace-node()" mode="internal:snapshot"
               as="namespace-node()" priority="3">
    <xsl:variable name="name" select="local-name(.)"/>
    <xsl:sequence select="internal:graft-to-parent(., ...
        function($n){$n/namespace-node()[local-name(.) = $name]} )"/>
</xsl:template>

<!-- make a copy C of a supplied node N, grafting it to a shallow copy of
     C's original parent, and returning the copy C -->

<xsl:function name="internal:graft-to-parent" as="node()">
    <xsl:param name="n" as="node()"/>
    <xsl:param name="original-parent" as="node()?" />
    <xsl:param name="down-function" as="function(node()) as node()"/>
    <xsl:choose>
        <xsl:when test="exists($original-parent)">
            <xsl:variable name="p" as="node()">
                <xsl:copy select="$original-parent">

```

```

        <xsl:copy-of select="@*"/>
        <xsl:copy-of select="$n"/>
    </xsl:copy>
</xsl:variable>
<xsl:variable name="copied-parent"
    select="internal:graft-to-parent(
        $p, $original-parent/..., function($n){$n/node()}))"/>
    <xsl:sequence select="$down-function($copied-parent)" />
</xsl:when>
<xsl:otherwise>
    <xsl:sequence select="$n"/>
</xsl:otherwise>
</xsl:choose>
</xsl:function>
```

Notes

The `snapshot` function is available for use (and is primarily intended for use) when a source document is processed using streaming. It can also be used when not streaming. The effect is to take a copy of the subtree rooted at the current node, along with copies of the ancestors and their attributes, and to make this available as a normal tree, that can be processed without any of the restrictions that apply while streaming, for example only being able to process children once. The copy, of course, does not include siblings of the context node or of its ancestors, so any attempt to navigate to these siblings will result in an empty sequence being returned.

For parentless nodes, the effect of `snapshot($x)` is identical to the effect of `copy-of($x)`.

Examples

Using `snapshot()` while streaming:

This example copies from the source document all employees who work in marketing and are based in Dubai. It assumes that employees are grouped by location. Because there are two accesses using the child axis (referencing `department` and `salary`), it is not possible to do this without buffering each employee in memory. The `snapshot` function is used in preference to the simpler `copy-of` so that access to attributes of the parent `location` element remains possible.

```

<xsl:source-document streamable="yes" href="employees.xml">
    <xsl:for-each select="snapshot(locations/location[@name='Dubai']
        /employee)[department='Marketing']">
        <employee>
            <location code="{$code}" />
            <salary value="{$salary}" />
        </employee>
    </xsl:for-each>
</xsl:source-document>
```

19 Streamability

This section contains rules that can be used to determine properties of [constructs](#) in the [stylesheet](#) — specifically, the [posture](#) and [sweep](#) of a construct — which enable the streamability of the stylesheet to be assessed.

These properties are used to determine the streamability of:

- [Template rules](#): see [6.6.4 Streamable Templates](#)
- The [xsl:source-document](#) instruction: see [18.1 The xsl:source-document Instruction](#)
- [Attribute sets](#): see [10.2.3 Streamability of Attribute Sets](#)
- [Accumulators](#): see [18.2.9 Streamability of Accumulators](#)
- [Stylesheet functions](#): see [19.8.5 Classifying Stylesheet Functions](#)
- The [xsl:merge](#) instruction: see [15.4 Streamable Merging](#)

In each case, the conditions for constructs to be [guaranteed-streamable](#) are defined in terms of these properties. The result of this analysis in turn (see [19.10 Streamability Guarantees](#)) imposes rules on how the constructs are handled by processors that implement the [streaming feature](#). The analysis has no effect on the behavior of processors that do not implement this feature.

The analysis is relevant to constructs such as streamable template rules and the [xsl:source-document](#) instruction that process a single streamed input document. The [xsl:merge](#) instruction, which processes multiple streamed inputs, has its own rules.

The rules in this section operate on the expression tree (more properly, construct tree) that is typically output by the XSLT and XPath parser. For the most part, the rules depend only on identifying the syntactic constructs that are present.

The rules in this section generally consider each [component](#) in the stylesheet (and in the case of [template rules](#), each template rule) in isolation. The exception is that where a component contains references to other components (such as global variables, functions, or named templates), then information from the signature of the referenced component is sometimes used. This is invariably information that cannot be changed if a component is overridden in a different [package](#). The analysis thus requires as a pre-condition that function calls and calls on named templates have been resolved to the extent that the corresponding function/template signature is known.

The detailed way in which the construct tree is derived from the lexical form of the stylesheet is not described in this specification. There are many ways in which the tree can be optimized without affecting the result of the rules in this section: for example, a sequence constructor containing a single instruction can be replaced by that instruction, and a parenthesized expression can be replaced by its content.

[**DEFINITION:** The term **construct** refers to the union of the following: a [sequence constructor](#), an [instruction](#), an [attribute set](#), a [value template](#), an [expression](#), or a [pattern](#).]

These [constructs](#) are classified into **construct kinds**: in particular, [instructions](#) are classified according to the name of the XSLT instruction, and [expressions](#) are classified according to the most specific production in the XPath grammar that the expression satisfies. (This means, for example, that $2+2$ is classified as an `AdditiveExpr`, rather than say as a `UnionExpr`; although it also satisfies the production rule for `UnionExpr`, `AdditiveExpr` is more specific.)

[**DEFINITION:** For every construct kind, there is a set of zero or more **operand roles**.] For example, an `AdditiveExpr` has two operand roles, referred to as the left-hand operand and the right-hand operand, while an `IfExpr` has three, referred to as the condition, the then-clause, and the else-clause. A function call with three

arguments has three operand roles, called the first, second, and third arguments. The names of the operand roles for each construct kind are not formally listed, but should be clear from the context.

[**DEFINITION:** In an actual instance of a construct, there will be a number of **operands**. Each operand is itself a **construct**; the construct tree can be defined as the transitive relation between constructs and their operands.] Each operand is associated with exactly one of the operand roles for the construct type. There may be operand roles where the operand is optional (for example, the `separator` attribute of the [xsl:value-of](#) instruction), and there may be operand roles that can be occupied by multiple operands (for example, the `xsl:when/@test` condition in [xsl:choose](#), or the arguments of the [concat](#)^{FO30} function).

Operand roles have a number of properties used in the analysis:

- The [required type](#) of the [operand](#). This is explicit in the case of function calls (the required type is defined in the function signature of the corresponding function). In other cases it is implicit in the detailed rules for the construct in question. In practice streamability analysis makes only modest use of the required type; the main case where it is relevant is for a function or template call, where knowing that the required type is atomic enables the inference that the [operand usage](#) for a supplied node is [absorption](#).
- [**DEFINITION:** The **operand usage**. This gives information, in the case where the operand value contains nodes, about how those nodes are used. The operand usage takes one of the values [absorption](#), [inspection](#), [transmission](#), or [navigation](#).] The meanings of these terms are explained in [19.3 Operand Roles](#). If the required type of the [operand](#) does not permit nodes to be supplied (for example because the required type is a function item or a map), then the operand usage is [inspection](#), because the only run-time operation on a supplied node will be to inspect it, discover it is a node, and raise a type error.

In the particular case where the required type is atomic, and any supplied nodes are atomized, the operand usage will be [absorption](#), because [atomize](#) is a special case of absorption.

- [**DEFINITION:** Whether or not the [operand](#) is **higher-order**. For this purpose an operand O of a construct C is higher-order if the semantics of C potentially require O to be evaluated more than once during a single evaluation of C .] More specifically, O is a **higher-order** operand of C if any of the following conditions is true:
 - The [context item](#) for evaluation of O is different from the context item for evaluation of C .
 - C is an [instruction](#) and O is a [pattern](#) (as with the `from` and `count` attributes of [xsl:number](#), and the `group-starting-with` and `group-ending-with` attributes of [xsl:for-each-group](#)).
 - C is an XPath `for`, `some`, or `every` expression and O is the expression in its `return` or `satisfies` clause.
 - C is an inline function declaration and O is the expression in its body.

Note:

There is one known case where this definition makes an operand higher-order even though it is only evaluated once: specifically, the sequence constructor contained in the body of an [xsl:copy](#) instruction that has a `select` attribute. See [19.8.4.12 Streamability of xsl:copy](#) for further details.

[**DEFINITION:** For some construct kinds, one or more operand roles may be defined to form a **choice operand group**. This concept is used where it is known that [operands](#) are mutually exclusive (for example the `then` and `else` clauses in a conditional expression).]

[**DEFINITION:** The **combined posture** of a [choice operand group](#) is determined by the [postures](#) of the [operands](#) in the group (the **operand postures**), and is the first of the following that applies:

1. If any of the operand postures is [roaming](#), then the combined posture is [roaming](#).
2. If all of the operand postures are [grounded](#), then the combined posture is [grounded](#).
3. If one or more of the operand postures is [climbing](#) and the remainder (if any) are [grounded](#), then the combined posture is [climbing](#).
4. If one or more of the operand postures is [striding](#) and the remainder (if any) are [grounded](#), then the combined posture is [striding](#).
5. If one or more of the operand postures is [crawling](#) and each of the remainder (if any) is either [striding](#) or [grounded](#), then the combined posture is [crawling](#).
6. Otherwise (for example, if the group includes both an operand with [climbing](#) posture and one with [crawling](#) posture), the combined posture is [roaming](#).

]

[**DEFINITION:** The **type-determined usage** of an [operand](#) is as follows: if the required type (ignoring occurrence indicator) is [function\(*\)](#) or a subtype thereof, then [inspection](#); if the required type (ignoring occurrence indicator) is an atomic or union type, then [absorption](#); otherwise [navigation](#).]

[**DEFINITION:** The **type-adjusted posture and sweep** of a construct C , with respect to a type T , are the [posture](#) and [sweep](#) established by applying the [general streamability rules](#) to a construct D whose single operand is the construct C , where the [operand usage](#) of C in D is the [type-determined usage](#) based on the required type T .]

Note:

In effect, the type-adjusted posture and sweep are the posture and sweep of the implicit expression formed to apply the [function conversion rules](#) to the argument of a function or template call, or to the result of a function or template, given knowledge of the required type. For example, an expression such as [discount](#) in the function call [abs\(discount\)](#), which would otherwise be [striding](#) and [consuming](#), becomes [grounded](#) and [consuming](#) because of the implicit atomization triggered by the function conversion rules.

The process of determining whether a construct is streamable reduces to determining properties of the constructs in the construct tree. The properties in question (which are described in greater detail in subsequent sections) are:

1. The **static type** of the construct. When the construct is evaluated, its value will always be an instance of this type. The value is a [U-type](#); although type inferencing is capable of determining information about the cardinality as well as the item type, the streamability analysis makes no use of this.
2. The **context item type**: that is, the static type of the [context item](#) potentially used as input to the construct. When the construct is evaluated, the context item used to evaluate the construct (if it is used at all) will be an instance of this type.
3. [**DEFINITION:** The **posture** of the expression. This captures information about the way in which the streamed input document is positioned on return from evaluating the construct. The posture takes one of the values [climbing](#), [striding](#), [crawling](#), [roaming](#), or [grounded](#).] The meanings of these terms are explained in [19.4 Determining the Posture of a Construct](#).

4. [DEFINITION: The **context posture**. This captures information about how the [context item](#) used as input to the construct is positioned relative to the streamed input. The **context posture** of a construct C is the posture of the expression whose value sets the focus for the evaluation of C.] Rules for determining the context posture of any construct are given in [19.5 Determining the Context Posture](#).

5. The **sweep** of the construct. The sweep of a construct gives information about whether and how the evaluation of the construct changes the current position in a streamed input document. The possible values are [motionless](#), [consuming](#), and [free-ranging](#). These terms are explained in [19.6 The Sweep of a Construct](#).

The values of these properties for a top-level construct such as the body of a template rule determine whether the construct is streamable.

The values of these properties are not independent. For example, if the static type is atomic, then the posture will always be grounded; if the sweep is free-ranging, then the posture will always be roaming.

The [posture](#) and [sweep](#) of a [construct](#), as defined above, are calculated in relation to a particular streamed input document. If there is more than one streamed input document, then a construct that is motionless with respect to one streamed input might be consuming with respect to another. In practice, though, the streamability analysis is only ever concerned with one particular streamed input at a time; constructs are analyzed in relation to the innermost containing [xsl:template](#), [xsl:source-document](#), [xsl:accumulator](#), or [xsl:merge-source](#) element, and this container implicitly defines the streamed input document that is relevant. The streamed input document affecting a construct is always the document that contains the context item for evaluation of that construct.

[19.1 Determining the Static Type of a Construct](#)

[DEFINITION: The **static type** of a [construct](#) is such that all values produced by evaluating the construct will conform to that type. The static type of a construct is a [U-type](#).]

[DEFINITION: A **U-type** is a set of [fundamental item types](#).]

[DEFINITION: There are 28 **fundamental item types**: the 7 node kinds defined in [\[XDM 3.0\]](#) (element, attribute, etc.), the 19 primitive atomic types defined in [\[XML Schema Part 2\]](#), plus the types `function(*)` and `xs:untypedAtomic`. The fundamental item types are disjoint, and every item is an instance of exactly one of them.]

More specifically, the fundamental item types are:

- `document-node()`, `element()`, `attribute()`, `text()`, `comment()`, `processing-instruction()`, `namespace-node()`;
- `xs:boolean`, `xs:double`, `xs:decimal`, `xs:float`, `xs:string`, `xs:dateTime`, `xs:date`, `xs:time`, `xs:gYear`, `xs:gYearMonth`, `xs:gMonth`, `xs:gMonthDay`, `xs:gDay`, `xs:anyURI`, `xs:QName`, `xs:NOTATION`, `xs:base64Binary`, `xs:hexBinary`, `xs:duration`
- `function(*)`
- `xs:untypedAtomic`

A value V (in general, a sequence) is an instance of a [U-type](#) U if every item in V is an instance of one of the [fundamental item types](#) in U . For example, the sequence (23, "Paris") is an instance of the U-type

$U\{xs:string, xs:decimal, xs:date\}$ because both items in the sequence belong to item types in this U-type.

Note:

It is a consequence of this rule that the empty sequence, (), is an instance of every U-type.

A [U-type](#) is represented in this specification using the notation $U\{t1, t2, t3, \dots\}$ where $t1, t2, t3, \dots$ are the names of the fundamental item types making up the U-type. The item types are represented using the syntax of the [ItemType^{XP30}](#) production in XPath, for example `comment()` or `xs:date`.

Note:

This means that the order of $t1, t2, t3, \dots$ has no significance: $U\{A, B\}$ is the same U-type as $U\{B, A\}$.

The smallest U-type is denoted $U\{\}$. This is not an empty type; like every other U-type, it has the empty sequence () as an instance. For convenience, the universal U-type is represented as $U\{*}$; the U-type corresponding to the set of 7 node kinds is written $U\{N\}$, and the U-type corresponding to all atomic values (that is, the 19 primitive atomic types plus `xs:untypedAtomic`) is written $U\{A\}$.

Because a [U-type](#) is a set, the operations of union, intersection, and difference are defined over U-types, and the result is always a U-type. If one U-type U is a subset of another U-type V , then U is said to be a subtype of V , and V is said to be a supertype of U .

In some cases the inference of a [static type](#) depends on the declared types of variables or functions. Since declared types use the [SequenceType](#) syntax, there is therefore a mapping defined from SequenceTypes to U-types. The mapping is as follows:

- The [SequenceType](#) `empty-sequence()` maps to $U\{\}$
- For every other [SequenceType](#), the mapping depends only on the item type and ignores the occurrence indicator. The mapping from item types is as follows:
 - `item()` maps to $U\{*}$
 - `AnyKindTest(node())` maps to $U\{N\}$
 - `DocumentTest` maps to $U\{document-node()\}$
 - `ElementTest` and `SchemaElementTest` map to $U\{element()\}$
 - `AttributeTest` and `SchemaAttributeTest` map to $U\{attribute()\}$
 - `TextTest` maps to $U\{text()\}$
 - `CommentTest` maps to $U\{comment()\}$
 - `PITest` maps to $U\{processing-instruction()\}$
 - `NamespaceNodeTest` maps to $U\{namespace-node()\}$
 - `FunctionTest`, `MapTest`, and (if the [XPath 3.1 Feature](#) is implemented) `ArrayTest` map to $U\{function(*)\}$
 - The QName `xs:error` maps to $U\{\}$

- A QName Q representing an atomic type that is a fundamental item type maps to $U\{Q\}$
- A QName Q representing an atomic type derived from a fundamental item type F maps to $U\{F\}$
- A QName Q representing a pure union type maps to a U-type containing the fundamental item types present in the transitive membership of the union, or from which the transitive members of the union are derived.

Although all constructs have a static type, the streamability analysis only needs to know the static type of XPath expressions, so the rules here are largely confined to that case. For patterns, the static type is deemed to be $U\{xs:boolean\}$, reflecting the fact that a pattern is essentially a function that can be applied to items to deliver a true or false (matching or non-matching) result. For constructs other than expressions and patterns, the static type for the purpose of streamability analysis is taken as $U\{*}$.

The rules given here are deliberately simple. Implementations may well be able to compute a more precise static type, but this will rarely be useful for streamability analysis. The item type for each kind of XPath expression is determined by the rules below. In the first column, numbers in square brackets are production numbers from the XPath 3.0 and XPath 3.1 specifications respectively. In the second column, the **Proforma** uses an informal notation used both to provide a reminder of the syntax of the construct in question, and to attach labels to its operand roles so that they can be referred to in the text of the third column.

Inferring a Static Type for XPath 3.0 Expressions

Construct	Proforma	Static Type
Expr [6,6]	E, F	the union of the static types of E and F
ForExpr [8,8]	for \$x in S return E	the static type of E
LetExpr [11,11]	let \$x := S return E	the static type of E
QuantifiedExpr [14,14]	some every \$x in S satisfies C	$U\{xs:boolean\}$
IfExpr [15,15]	if (C) then T else E	the union of the static types of T and E
OrExpr [16,16]	E or F	$U\{xs:boolean\}$
AndExpr [17,17]	E and F	$U\{xs:boolean\}$
ComparisonExpr [18,18]	E = F, E eq F, E is F	$U\{xs:boolean\}$
StringConcatExpr [19,19]	E F	$U\{xs:string\}$
RangeExpr [20,20]	E to F	$U\{xs:decimal\}$

Construct	Proforma	Static Type
AdditiveExpr [21,21]	E + F	$U\{A\}$. But if the expression is a predicate (that is, if it appears between square brackets in a filter expression or axis step), then $U\{xs:decimal, xs:double, xs:float\}$
MultiplicativeExpr [22,22]	E * F	$U\{A\}$. But if the expression is a predicate (that is, if it appears between square brackets in a filter expression or axis step), then $U\{xs:decimal, xs:double, xs:float\}$
UnionExpr [23,23]	E F	the union of the static types of E and F
IntersectExceptExpr [24,24]	E intersect F E except F	the intersection of the static types of E and F the static type of E
InstanceOfExpr [25,25]	E instance of T	$U\{xs:boolean\}$
TreatExpr [26,26]	E treat as T	the U-type corresponding to the SequenceType T
CastableExpr [27,27]	E castable as T	$U\{xs:boolean\}$
CastExpr [28,28]	E cast as T	if T is an atomic or pure union type, the corresponding U-type. Otherwise, for example if T is a list type, $U\{A\}$.
UnaryExpr [29,30]	-N	$U\{xs:decimal, xs:double, xs:float\}$
SimpleMapExpr [34,35]	E ! F	the static type of F
PathExpr [35,36]	/ /P //P	$U\{document-node()\}$ the static type of P the static type of P
RelativePathExpr [36,37]	P/Q, P//Q	the static type of Q
AxisStep [38,39]	E[P]	the static type of E: see 19.1.1 Static Type of an Axis Step
ForwardStep [39,40], ReverseStep [42,43]	Axis::NodeTest	See 19.1.1 Static Type of an Axis Step
PostfixExpr [48,49]	Filter Expression E[P] Dynamic Function Call F(X, Y)	the static type of E $U\{*\}$, unless ancillary information is available about the function signature of F: see below.

Construct	Proforma	Static Type
Literal [53,57]	"pH", 93.7	$U\{xs:string\}$, $U\{xs:decimal\}$, or $U\{xs:double\}$, depending on the form of the literal
VarRef [55,59]	\$V	For a variable declared using xsl:variable or xsl:param , and for parameters of inline function expressions: the declared type of the variable, defaulting to $U\{*\}$. For variables declared using <code>for</code> , <code>let</code> , <code>some</code> , and <code>every</code> expressions: the static type of the expression to which the variable is bound.
ParenthesizedExpr [57,61]	(E)	the type of E
	()	$U\{\}$ (a type whose only instance is the empty sequence)
ContextItemExpr [58,62]	.	the context item type: see below
FunctionCall [59,63]	F(X, Y)	In general: the U-type corresponding to the declared result type of function F. But: <ul style="list-style-type: none"> If one or more of the arguments to the function have operand usage transmission, then the intersection of the U-type corresponding to the declared result type with the union of the static types of the arguments having usage transmission. (For example, the static type of the function call <code>head(//text())</code> is $U\{text()\}$.) Special rules apply to the current function: see 19.1.2 Static Type of a Call to current.
NamedFunctionRef [63,67]	F#n	$U\{function(*)\}$
InlineFunctionExpr [64,68]	function(P) {E}	$U\{function(*)\}$
MapConstructor [-,69]	map{"A":E, "B":F}	$U\{function(*)\}$
Postfix Lookup [-,49]	E ? K	If the type of E is a map type <code>map(K, V)</code> or an array type <code>array(V)</code> , then the U-type corresponding to the item type of V; otherwise $U\{*\}$
(Unary) Lookup [-,53]	? K	If the context item type is a map type <code>map(K, V)</code> or an array type <code>array(V)</code> , then the U-type corresponding to the item type of V; otherwise $U\{*\}$

Construct	Proforma	Static Type
ArrowExpr [-,29]	$X \Rightarrow F(Y, Z)$	The static type of the equivalent static or dynamic function call $F(X, Y, Z)$
SquareArrayConstructor [-,74]	$[X, Y, \dots]$	$U\{function(*)\}$
CurlyArrayConstructor [-,75]	$\text{array}\{X, Y, \dots\}$	$U\{function(*)\}$

Where the [static type](#) of an expression is $U\{function(*)\}$, it is useful to retain additional information: specifically, the signature of the function. This may be regarded as information ancillary to the U-type of the expression; it does not play any role in operations such as testing whether one U-type is a subtype of another, or forming the union of two U-types. This ancillary information is available for a `NamedFunctionRef`, for an `InlineFunctionExpr`, for a `MapConstructor`, for a `FunctionCall` whose static type is $U\{function(*)\}$, and for a `VarRef` if the variable is bound to any of the forgoing, or if it has a declared type corresponding to $U\{function(*)\}$.

Note:

The special case type inference used for an `AdditiveExpr` or `MultiplicativeExpr` appearing as a predicate is possible because if an arithmetic operation within a predicate produces any other result, for example an `xs:duration` or `xs:dateType`, this would cause a type error (on the grounds that an `xs:duration` or `xs:dateType` has no effective boolean value), and static type inference only needs to consider the type of non-error results. The benefit of this special rule is that filter expressions such as `/descendant::section[$i + 1]` can be recognized as returning a singleton, and therefore as being [striding](#), even if the type of `$i` is unknown.

19.1.1 Static Type of an Axis Step

An `AxisStep` consists of either a `ForwardStep` or `ReverseStep` followed by zero or more predicates. The predicates have no effect on the inferred type of the `AxisStep`.

The static type of an abbreviated step is the static type of its expansion, for example the static type of `@*` is the same as the static type of `attribute::*`.

Both the constructs `ForwardStep` or `ReverseStep`, in their unabbreviated form, are written as `Axis::NodeTest`. The static type depends on both the `Axis` and the `NodeTest`, and also on the [context item type](#), determined as described in [19.2 Determining the Context Item Type](#).

If the [context item type](#) has an empty intersection with $U\{N\}$ (that is, if the context item type cannot be a node), then evaluation of the `AxisStep` will always fail; it is permissible to raise a type error statically in this case, but for the sake of the analysis, the static type of the `AxisStep` can be taken as $U\{\}$. In other cases, let `CIT` be the intersection of the [context item type](#) with $U\{N\}$.

Let $K(A, CIT)$ be the set of **reachable node kinds** given an axis A (a [U-type](#)) as defined by the following table:

Axis	Reachable Node Kinds

self	<i>CIT</i>
attribute	if <i>CIT</i> includes $U\{\text{element}()\}$ then $U\{\text{attribute}()\}$ else $U\{\}$
namespace	if <i>CIT</i> includes $U\{\text{element}()\}$ then $U\{\text{namespace-node}()\}$ else $U\{\}$
child, descendant	if <i>CIT</i> includes $U\{\text{element}()\}$ or $U\{\text{document-node}()\}$ then $U\{\text{element}(), \text{text}(), \text{comment}(), \text{processing-instruction}()\}$ else $U\{\}$
following-sibling, preceding-sibling, following, preceding	if <i>CIT</i> is $U\{\text{document-node}()\}$ then $U\{\}$ else $U\{\text{element}(), \text{text}(), \text{comment}(), \text{processing-instruction}()\}$
parent, ancestor	if <i>CIT</i> is $U\{\text{document-node}()\}$ then $U\{\}$ else $U\{\text{element}(), \text{document-node}()\}$
ancestor-or-self	the union of $K(\text{ancestor}, \text{CIT})$ and <i>CIT</i>
descendant-or-self	the union of $K(\text{descendant}, \text{CIT})$ and <i>CIT</i>

Let $T(NT)$ be the set of node kinds that are capable of satisfying a **NodeTest** NT , defined by the following table:

NodeTest	Possible Node Kinds
AnyKindTest (that is, <code>node()</code>)	$U\{N\}$ (that is, any node)
Any other KindTest	The corresponding U-type (for example, $U\{\text{text}()\}$ for the KindTest <code>text()</code>)
NameTest	The U-type corresponding to the principal node kind of the specified axis

The static type of an **AxisStep** with axis A and node test NT , given a context item type CIT , is then defined to be the intersection of $K(A, CIT)$ with $T(NT)$.

19.1.2 [Static Type of a Call to current](#)

The rules in this section define the static type of a call to the [current](#) function.

1. If the call is within a [pattern](#), the static type of the function call is the [match type](#) of the pattern.

Note:

There is no circularity in this definition: a call to [current](#) in a pattern can only appear within a predicate, and the match type of a pattern never depends on anything appearing in a predicate.

2. Otherwise (the function call is within an XPath expression), the static type of the function call is the [context item type](#) that applies to the outermost containing XPath expression, determined by the rules in [19.2 Determining the Context Item Type](#).

19.1.3 [Schema-Aware Streamability Analysis](#)

Note:

The streamability analysis in this chapter is not schema-aware. There are cases where use of schema type information might enable a processor to determine that a construct is streamable when it would be unable to make this determination otherwise. Two examples:

- A processor might decide that a construct such as `price + salesTax` is streamable if both the child elements have a simple type such as `xs:decimal`, or if the order in which they appear in the input document is known.
- A processor might decide that a step using the descendant axis, such as `./title`, has striding rather than crawling posture if it can establish that two `title` elements will never be nested (that is, a `title` cannot contain another `title`). This would allow the instruction `<xsl:apply-templates select=".//title"/>` to be used in a streaming template rule.

Although such constructs are not guaranteed streamable according to this specification, there is nothing to prevent a processor providing a streamed implementation if it is able to do so.

[19.2 Determining the Context Item Type](#)

[**DEFINITION:** For every expression, it is possible to establish by static analysis, information about the item type of the context item for evaluation of that expression. This is called the **context item type** of the expression.]

The context item type of an expression is a U-type.

The semantics of every construct, defined in this specification or in the XPath specification, describe how the focus for evaluating each operand of the construct is determined. In most cases the focus is the same as that of the parent construct. In some cases the focus is determined by evaluating some other expression, for example in the expressions A/B , $A!B$, or $A[B]$, the focus for evaluating B is A . More generally:

- [**DEFINITION:** A **focus-changing construct** is a construct that has one or more operands that are evaluated with a different focus from the parent construct.]

Note:

Examples of focus-changing constructs include the instructions `xsl:for-each`, `xsl:iterate`, and `xsl:for-each-group`; path expressions, filter expressions, and simple mapping expressions; and all patterns.

- [**DEFINITION:** Within a focus-changing construct there is in many cases one operand whose value determines the focus for evaluating other operands; this is referred to as the **controlling operand**.]

Note:

For example, the controlling operand of an `xsl:for-each`, `xsl:iterate`, or `xsl:for-each-group` instruction is the expression in its `select` attribute; the controlling operand of a filter expression $E[P]$ is E , and the controlling operand of a simple mapping expression $A!B$ is A .

- [**DEFINITION:** Within a focus-changing construct there are one or more operands that are evaluated with a focus determined by the controlling operand (or in some cases such as `xsl:on-completion`, with an absent focus);

these are referred to as **controlled operands**.]

Note:

For example, the main controlled operand of an [xsl:for-each](#), [xsl:iterate](#), or [xsl:for-each-group](#) instruction is the contained sequence constructor; the controlled operand of a filter expression $E[P]$ is P , and the controlled operand of a simple mapping expression $A!B$ is B .

- [**DEFINITION:** The **focus-setting container** of a construct C is the innermost **focus-changing construct** F (if one exists) such that C is directly or indirectly contained in a **controlled operand** of F . If there is no such construct F , then the focus-setting container is the containing **declaration**, for example an [xsl:function](#) or [xsl:template](#) element.]

Note:

For example, if an instruction appears as a child of [xsl:for-each](#), then its focus-setting container is the [xsl:for-each](#) instruction; if an expression appears within the predicate of a filter expression, its focus-setting container is the filter expression.

The [context item type](#) of a construct C is the first of the following that applies:

- If the **focus-setting container** of C is an [xsl:function](#) element, an inline function declaration, or an [xsl:on-completion](#) element, then the context item type is $U\{\}$.
- If the **focus-setting container** of C is an [xsl:source-document](#) instruction, then the context item type is $U\{document-node()\}$.
- If the **focus-setting container** of C is a [template rule](#), then the context item type is the [match type](#) of the match pattern of the template rule, defined below.
- If the **focus-setting container** of C is a [PredicatePattern](#), then the context item type is $U\{*}$.
- If the **focus-setting container** is a [global variable](#) declaration, the context item type is determined by the [type](#) attribute of the [xsl:global-context-item](#) declaration, defaulting to $U\{*}$, or $U\{\}$ if the [xsl:global-context-item](#) declaration specifies [use="absent"](#).
- If the **focus-setting container** is any other [declaration](#), for example [xsl:key](#) or [xsl:accumulator](#), the context item type is $U\{*}$.
- Otherwise, the context item type is the [static type](#) (see [19.1 Determining the Static Type of a Construct](#)) of the [controlling operand](#) of the **focus-setting container** of C .

[**DEFINITION:** The **match type** of a [pattern](#) is the most specific [U-type](#) that is known to match all items that the pattern can match.] The match type of a pattern is the inferred [static type](#) of the pattern's equivalent expression, determined according to the rules in [19.1 Determining the Static Type of a Construct](#). For example, the match type of the pattern `para[1]` is $U\{element()\}$, while that of the pattern `@code[.= 'x']` is $U\{attribute()\}$

19.3 Operand Roles

An [operand role](#) gives information about the [operands](#) of a particular kind of construct. The two important properties of an operand role are the required type and the operand usage.

The [usage](#) of an operand role is relevant only when the value of an [operand](#) supplied in that role is a node, or a sequence that contains nodes. It is one of the following:

- [DEFINITION: An operand usage of **absorption** indicates that the construct reads the subtree(s) rooted at a supplied node(s).] Examples are constructs that atomize their [operands](#), or that obtain the string value of a supplied node, or that copy the supplied node to a new tree. Another example is the [deep-equal^{FO30}](#) function, which compares the subtrees rooted at the nodes supplied in its first two arguments.
- [DEFINITION: An operand usage of **inspection** indicates that the construct accesses properties of a supplied node that are available without reading its subtree.] Examples are functions such as [name^{FO30}](#) and [base-uri^{FO30}](#), and the [instance of](#) expression which tests the type of a node (or other item), or functions such as [count^{FO30}](#), [exists^{FO30}](#), and [boolean^{FO30}](#) which are only interested in the existence of the node, and not in its properties.
- [DEFINITION: An operand usage of **transmission** indicates that the construct will (potentially) return a supplied node as part of its result to the calling construct (that is, to its parent in the construct tree).] It also indicates that document order is preserved: if the input is in document order, then the result must be in document order. An example is a filter expression, where nodes in the base expression (the expression being filtered) will typically appear in the result of the filter expression, in their original order.
- [DEFINITION: An operand usage of **navigation** indicates that the construct may navigate freely from the supplied node to other nodes in the same tree, in a way that is not constrained by the streamability rules.] This covers several cases: cases where it is known that the construct performs impermissible navigation (for example, the [xsl:number](#) instruction) or reordering (the [reverse^{FO30}](#) function), or that require look-ahead (the [innermost^{FO30}](#) function) and also cases where the analysis is unable to determine what use is made of the node, for example because it is passed as an argument to a user-defined function, or retained in a variable.

The concept of operand usage is not used for all constructs (for example, it is not used in the analysis of path expressions). Where it is used, the assignment of operand usages to each operand role of a construct is defined in [19.8 Classifying Constructs](#).

19.3.1 Examples showing the Effect of Operand Usage

Example: The Effect of Operand Usage on the Streamability of a Context Item Expression

Consider the following construct:

```
<xsl:source-document streamable="yes" href="emps.xml">
  <xsl:for-each select="*/emp">
    <xsl:value-of select=". />
  </xsl:for-each>
</xsl:source-document>
```

To assess the streamability, we follow the following logic:

1. The top-level construct is a sequence constructor. It is evaluated with a document node as the context item, and with a striding posture.
2. The sequence constructor has one child instruction, which has an operand usage of transmission.
3. The xsl:for-each instruction evaluates its select expression, with the context item and posture unchanged.
4. The step child::* is evaluated with this context item and posture. The posture transition rules permit this; we now have a sequence of child elements, and still a striding posture.
5. The same applies to the next step, child::emp
6. The content of the xsl:for-each instruction is a sequence constructor which itself has a single operand, the xsl:value-of instruction.
7. The xsl:value-of instruction is evaluated once for each emp child, with that child as context item and in a striding posture. This instruction uses the general streamability rules. The operand usage of the select expression is absorption. This means that the result of the xsl:value-of instruction is grounded and consuming.
8. The result of the trivial sequence constructor contained in the xsl:for-each instruction is therefore grounded and consuming.
9. The result of the xsl:for-each instruction (see [19.8.4.18 Streamability of xsl:for-each](#)) is therefore grounded and consuming.
10. The result of the trivial sequence constructor contained in the xsl:source-document instruction is therefore grounded and consuming.
11. The xsl:source-document instruction is therefore guaranteed-streamable.

Now consider a slightly different construct:

```
<xsl:source-document streamable="yes" href="emps.xml">
  <xsl:for-each select="*/emp">
    <xsl:sequence select=". />
  </xsl:for-each>
</xsl:source-document>
```

To assess the streamability, we follow the following logic:

1. The top-level construct is a sequence constructor. It is evaluated with a document node as the context item, and with a striding posture.
2. The sequence constructor has one child instruction, which has an operand usage of transmission.

3. The [xsl:for-each](#) instruction evaluates its `select` expression, with the context item and [posture](#) unchanged.
4. The step `child::*` is evaluated with this context item and posture. The posture transition rules permit this; we now have a sequence of child elements, and still a [striding](#) posture.
5. The same applies to the next step, `child::emp`
6. The content of the [xsl:for-each](#) instruction is a [sequence constructor](#) which itself has a single operand, the [xsl:sequence](#) instruction.
7. The [xsl:sequence](#) instruction is evaluated once for each `emp` child, with that child as context item and in a [striding](#) posture. This instruction uses the [general streamability rules](#). The [operand usage](#) of the `select` expression is [transmission](#). This means that the result of the [xsl:sequence](#) instruction is [striding](#) and [motionless](#).
8. The result of the trivial sequence constructor contained in the [xsl:for-each](#) instruction is therefore also [striding](#) and [motionless](#).
9. The result of the [xsl:for-each](#) instruction (see [19.8.4.18 Streamability of xsl:for-each](#)) is therefore [striding](#) and [consuming](#) (the wider of the sweeps of the `select` expression and the sequence constructor).
10. The result of the trivial sequence constructor contained in the [xsl:source-document](#) instruction is therefore [striding](#) and [consuming](#).
11. Since the result is not [grounded](#), the [xsl:source-document](#) instruction is therefore not [guaranteed-streamable](#).

Expressed informally, the result of a [declared-streamable xsl:source-document](#) instruction (or of a [declared-streamable](#) template rule) must not contain streamed nodes. The reason for this is that once streamed nodes are returned to constructs that are not declared streamable and therefore have no streamability constraints, there is no way to analyze what happens to them, and thus to guarantee streamability.

Example: The Effect of Operand Roles on the Streamability of Path Expressions

Consider the expression `./chapter`.

When this appears as an argument to the function `countFO30` or `existsFO30`, it can be streamed (it is a consuming expression, meaning that the subtree rooted at the context item needs to be read in order to evaluate the expression). A possible strategy for performing a streamed evaluation is to read all descendants of the context item in document order, checking each one to see whether its name is `chapter`. The sweep of the expression will be consuming, and its posture will be crawling.

The operand usage (the usage of the argument to `countFO30` or `existsFO30`) is defined as inspection. The general streamability rules show that when the posture of an operand is crawling and the operand usage is inspection, the resulting expression is grounded and consuming. This means that (in the absence of other consuming expressions) the containing template or function will generally be streamable.

In the expression `tail(./chapter)`, the operand usage is classified as transmission, meaning that the nodes are simply passed up the tree to the next containing expression. In general, when a crawling expression is passed as an argument and the operand role is transmission, the containing expression will also be crawling. However, there is an exception where the expression is known to deliver a singleton (for example, `head(./chapter)`). In this case the returned sequence cannot contain any nested nodes, so it is crawling.

When the same expression appears as an argument to an atomizing function `string-joinFO30`, the processor knows that it will need to access the subtree of each selected `section` element in order to compute the result of the function (the argument to `string-joinFO30` is classified as having operand usage absorption). The processor does not know whether these subtrees will be nested (one `section` might contain another). In most cases they will not be nested, because atomizing a sequence that contains nested nodes is not generally a useful thing to do. The streamability analysis therefore makes an optimistic assumption, by treating atomization of a crawling expression as a streamable operation. In the worst case, where it turns out that the selected nodes are indeed nested, the processor must handle this, typically by buffering the content of inner nodes until the end tag of the outer nodes is reached.

This treatment of nodes in a crawling expression applies to all cases in which the content of the nodes is handled in a way defined entirely by the rules of this specification: for example, operations such as atomization, obtaining the string value of nodes, deep copy of nodes, and the `deep-equalFO30` function. It does not extend to cases where the processing applied to the nodes is user-defined: for example, operations such as `xsl:apply-templates`, `xsl:for-each`, or `xsl:for-each-group`. In these cases, the nodes selected for processing must not be nested (a crawling posture is not permitted in these contexts).

When a crawling expression appears as an argument to a call on a user-defined function, the effect depends on the streamability category of the function, as described in [19.8.5 Classifying Stylesheet Functions](#).

19.4 Determining the Posture of a Construct

The **posture** of a construct indicates the relationship of the nodes selected by the construct to a streamed input document. The value is one of the following:

- [DEFINITION: **Grounded**: indicates that the value returned by the construct does not contain nodes from the streamed input document]. Atomic values and function items are always grounded; nodes are grounded if it is

known that they are in a non-streamed document. For example the expressions `doc('x')` and `copy-of(.)` both return grounded nodes.

- [DEFINITION: **Climbing**: indicates that streamed nodes returned by the construct are reached by navigating the parent, ancestor[-or-self], attribute, and/or namespace axes from the node at the current streaming position.] When the `context posture` is climbing, use of certain axes such as `parent` and `ancestor` is permitted, but use of other axes such as `child` or `descendant` violates the streamability rules.
- [DEFINITION: **Crawling**: typically indicates that streamed nodes returned by a construct are reached by navigating the descendant[-or-self] axis.] Nodes reached in this way are potentially nested (one might be an ancestor of another), so further downward navigation is not permitted. Expressions that can be statically determined to return a singleton node (for example `head(./title)`) generate a result with no such nesting, so they are striding rather than crawling.
- [DEFINITION: **Striding**: indicates that the result of a construct contains a sequence of streamed nodes, in document order, that are peers in the sense that none of them is an ancestor or descendant of any other.] This is typically achieved by using one or more steps involving the `child` or `attribute` axes only. Use of the `outermost`^{FO30} function can also result in a striding posture, as can functions such as `head`^{FO30} or `zero-or-one`^{FO30} that ensure the result will be a singleton node.
- [DEFINITION: **Roaming**: indicates that the nodes returned by an expression could be anywhere in the tree, which inevitably means that the construct cannot be evaluated using streaming.] For example, the `posture` of an axis step using the `following` or `preceding` axis will typically be `roaming`, which leads the analysis to conclude that the construct is not streamable.

Note:

One way to think about the posture values is as labels for states in a finite state automaton, where the alphabet of symbols accepted by the automaton is the set of 13 XPath axes, and the sentence being parsed is a path expression containing a sequence of axis steps. For example, use of the `descendant` axis when the current state is **striding** moves the new state to **crawling**, and use of the `parent` axis then takes it to **climbing**.

The `posture` of a construct is determined in one of several ways:

- For axis steps, the posture of the expression is determined by the `context posture` and the choice of axis. For example, an axis step using the `ancestor` axis always has a posture of `climbing`, while an axis step using the `child` axis, if the `context posture` is `striding`, will itself have a posture of `striding`. The rules for the posture transitions produced by axis steps are given in [19.8.8.9 Streamability of Axis Steps](#).
- For many other constructs, the posture is determined by the `general streamability rules`. These determine the result posture in terms of the `operands` of the construct and the way in which each operand is used. For example, a construct that accepts a streamed node as the value of an operand, and atomizes that node, will generally have a posture of `grounded`.
- Other constructs have their own special rules, which are all listed in this chapter. For example, a call on the `root`^{FO30} function behaves analogously to an axis step, and is described in [19.8.9.18 Streamability of the root Function](#). Special rules are needed for:
 - Constructs that evaluate an `operand` more than once, such as an XPath `for` expression;
 - Constructs that have alternatives among their operands, such as an XPath `if` expression;
 - Constructs that navigate relative to the context item, such as axis steps;

- Constructs with implicit inputs, such as the context item expression . (dot);
- Constructs that change the focus, such as a filter expression;
- Constructs that invoke functions or templates.

The characterization of an expression as striding, crawling, climbing, or roaming applies only to the streamed nodes in the result of the expression. The result of the expression may also contain non-streamed (grounded) nodes or atomic values. For example if $/x/y$ is a striding expression, then $(/x/y \mid \$doc//x)$ is also striding, given that $\$doc$ contains non-streamed nodes. The assertion that the nodes in the result of a striding expression are in document order and are peers thus applies only to the subset of the nodes that are streamed.

Note:

A consequence of this is that when striding expressions are used in a context that requires sorting into document order, for example $(/x/y \mid \$doc//x) / @price$, the fact that the expression is striding does not eliminate the need for the sequence to be re-ordered. However, there will never be a need for the relative order of the streamed nodes in the value to change.

Since the data model leaves the relative order of nodes in different trees implementation-defined, and since streamed and unstreamed nodes will necessarily be in different trees, a useful implementation strategy might be to arrange that streamed nodes always precede unstreamed nodes in document order (or vice versa). An operation that needs to process the result of a striding expression in document order can then first deliver all the streamed nodes (by consuming the input stream) in the order they arrive, and then deliver the unstreamed nodes, suitably sorted.

19.5 Determining the Context Posture

In the same way as the type of the context item can be determined for any construct C by reference to the type of the construct that establishes the context for the evaluation of C , so the posture of the context item C can be determined by reference to the posture of the construct that establishes the context.

The context posture of a construct C is the first of the following that applies:

1. If the focus-setting container of C is an xsl:function declaration, an inline function declaration, or an xsl:on-completion element, then the context posture is roaming.

Note:

This is essentially an error case; expressions that depend on the context item should not normally appear within these constructs.

2. If the focus-setting container of C is an xsl:source-document instruction, then the context posture is striding if the instruction is declared-streamable, or grounded otherwise.
3. If the focus-setting container of C is a template rule whose mode is declared with streamable="yes", then the context posture is striding.
4. If the focus-setting container of C is a pattern, then the context posture is striding.
5. If the focus-setting container of C is an xsl:attribute-set declaration with the attribute streamable="yes", then the context posture is striding.

6. If the focus-setting container is any other declaration, for example a global variable declaration, a named template, or a template rule or attribute set that does not specify `streamable="yes"`, then the context posture is roaming.
7. Otherwise, the context posture is the posture of the controlling operand of the focus-setting container of C.

19.6 The Sweep of a Construct

[DEFINITION: Every construct has a **sweep**, which is a measure of the extent to which the current position in the input stream moves during the evaluation of the expression. The sweep is one of: motionless, consuming, or free-ranging.] This list of values is ordered: a free-ranging expression has **wider sweep** than a consuming expression, which has **wider sweep** than a motionless expression.

[DEFINITION: A **motionless** construct is any construct deemed motionless by the rules in this section ([19 Streamability](#)).] Informally, a motionless construct is one that can be evaluated without changing the current position in the input stream.

Note:

The context item expression `.` is classified as motionless; however a construct that uses `.` as an operand (for example, `string(.)`) might be consuming. The streamability rules effectively consider expressions such as `.` within the context of the containing construct.

[DEFINITION: A **consuming** construct is any construct deemed consuming by the rules in this section ([19 Streamability](#)).] Informally, a consuming construct is one whose evaluation requires repositioning of the input stream from the start of the current node to the end of the current node.

[DEFINITION: A **free-ranging** construct is any construct deemed free-ranging by the rules in this section ([19 Streamability](#)).] Informally, a free-ranging construct is one whose evaluation may require access to information that is not available from the subtree rooted at the current node, together with information about ancestors of the current node and their attributes.

The table below shows some examples of expressions having different combinations of posture and sweep.

Combinations of Sweep and Posture

	Motionless	Consuming	Free-Ranging
Grounded	<code>name()</code>	<code>string(title)</code>	See Note
Climbing	<code>parent::*</code>	<code>child::x/ancestor::y</code>	See Note
Striding	<code>@status</code>	<code>child::*</code>	See Note
Crawling	The subexpression <code>.</code> in <code>//a/.</code>	<code>descendant::*</code>	<code>//x[child::y]</code>
Roaming	See Note	See Note	<code>preceding::*</code>

Note:

In all cases where either the [posture](#) is [roaming](#), or the [sweep](#) is [free-ranging](#), or both, the effect is to make an expression non-streamable. For convenience, therefore, evaluation of the streamability rules in most cases returns the values [roaming](#) and [free-ranging](#) only in combination with each other. In cases where the rules return a [posture](#) of [roaming](#) combined with some other [sweep](#), or a [sweep](#) of [free-ranging](#) with some other [posture](#), the final result of the analysis is always the same as if the expression were both [roaming](#) and [free-ranging](#).

For an example of a case where an expression is [roaming](#) but not [free-ranging](#), consider the right-hand operand of the relative path expression (`preceding::x/. .`). The rules for the streamability of a context item expression (see [19.8.8.13 Streamability of the Context Item Expression](#)) give "`.`" in this context a [roaming](#) posture, combined with [motionless](#) sweep. But the relative path expression as a whole is [roaming](#) and [free-ranging](#) (see [19.8.8.8 Streamability of Path Expressions](#)), so the apparent inconsistency is transient.

[19.7 Grounded Consuming Constructs](#)

A construct is grounded if the items it delivers do not include nodes from a streamed document; it is consuming if evaluation of the construct reads nodes from a streamed input in a way that requires advancing the current position in the input.

Grounded consuming constructs play an important role in streaming, and this section discusses some of their characteristics.

Examples of grounded consuming constructs (assuming the context item is a streamed node) include:

- `sum(./transaction/@value)`
- `copy-of(./account/history/event)`
- `distinct-values(./account/@account-nr)`
- `<xsl:for-each select="transaction"><t><xsl:value-of select="@value"/></t></xsl:for-each>`

XSLT 3.0 provides the two functions [copy-of](#) and [snapshot](#) with the explicit purpose of creating a sequence of grounded nodes, that can be processed one-by-one without the usual restrictions that apply to streamed processing, such as the rule permitting at most one downward selection. The processing style that exploits these functions is often called “windowed streaming”.

In general the result of a grounded consuming construct is a sequence. Depending on how this sequence is used, it may or may not be necessary for the processor to allocate sufficient memory to hold the entire sequence. The streamability rules in this specification place few constraints on how a grounded sequence is used. This is deliberate, because it gives users control: by creating a grounded sequence (for example, by use of the [copy-of](#) function) stylesheet authors create the possibility to process data in arbitrary ways (for example, by sorting the sequence), and accept the possibility that this may consume memory.

Pipelined evaluation of a sequence is analogous to streamed processing of a source document. Pipelined evaluation occurs when the items in a sequence can be processed one-by-one, without materializing the entire sequence in

memory. Pipelining is a common optimization technique in all functional programming languages. Operations for which pipelined evaluation is commonly performed include filtering (`$transactions[@value gt 1000]`), mapping (`$transactions ! (@value - @processing-fee)`), and aggregation (`sum($transactions)`). Operations that cannot be pipelined (because, for example, the first item in the result sequence cannot be computed without knowing the last item in the input sequence) include those that change the order of items (`reverse()`, `sort()`). Other operations such as `distinct-values()` allow the input to be processed one item at a time, but require memory that potentially increases as the sequence length increases. Saving a grounded sequence in a variable is also likely in many cases to require allocation of memory to hold the entire sequence.

When the input to an operation is a grounded consuming sequence (more accurately, a sequence resulting from the evaluation of a grounded consuming construct), this specification does not attempt to dictate whether the operation is pipelined or not. The goal of interoperable streaming in finite memory can therefore only be achieved if stylesheet authors take care to avoid constructing grounded sequences that occupy large amounts of memory. In practice, however, users can expect that many grounded consuming constructs will be pipelined where the semantics permit this.

Note:

Some processors may recognize an opportunity for pipelining only if the expression is written in a particular way. For example the constructs `copy-of(/a/b/c)` and `/a/b/c/copy-of(.)` are to all intents and purposes equivalent, but some processors might recognize the second form more easily as suitable for pipelining.

(There is one minor difference between these expressions: the order of nodes in `copy-of(/a/b/c)` is required to reflect the document order of the nodes in `/a/b/c`, while the result of `/a/b/c/copy-of(.)` can be in any order, in consequence of the rule that document order for nodes in different trees is implementation-dependent.)

The use of the `lastFO30` function requires particular care because of its effect on pipelining. The streamability rules prevent the use of `last()` in conjunction with an expression that returns streamed nodes (because it would require look-ahead in the stream), but there is no similar constraint for grounded sequences. So for example it is not permitted (in a context that requires streaming) to write

```
<xsl:for-each select="transaction">
  <xsl:value-of select="position(), ' of ', last()"/>
</xsl:for-each>
```

but it is quite permissible to write

```
<xsl:for-each select="transaction/copy-of()">
  <xsl:value-of select="position(), ' of ', last()"/>
</xsl:for-each>
```

because the call on `copy-of` makes the sequence grounded. This construct cannot be pipelined because computing the first item in the result sequence depends on knowing the length of the input sequence; in consequence, a processor might be obliged to buffer all the transactions (or their copies) in memory. In this simple example the impact of the call on `lastFO30` is easily detected both by the human reader and by the XSLT processor, but there are other cases where the effect is less obvious. For example if the stylesheet executes the instruction

```
<xsl:apply-templates select="transaction/copy-of(.)"/>
```

then the presence of a call on `last`^{FO30} in one of the template rules that gets invoked might not be easily spotted; yet the effect is exactly the same in preventing the result being computed by processing input items strictly one at a time. Avoiding such effects is entirely the responsibility of the stylesheet author.

By contrast, there is no intrinsic reason why use of the `position`^{FO30} should prevent pipelined processing: all it requires is for the processor to count how many items have been processed so far. Processors may also be able to handle the construct `position() = last()` without storing the entire sequence in memory; rather than actually evaluating the numeric values of `position()` and `last()`, this can be done by testing whether the context item is the last item in the sequence, which only requires a one-item lookahead.

19.8 Classifying Constructs

This section defines the properties of every kind of `construct` that may appear in a `stylesheet`. It identifies the `operand roles` and their `usage`, and it gives the rules that define the `posture` and `sweep` of the construct. In cases where the `general streamability rules` apply, there is still an entry for the construct in order to define its `operands` and their usages, since this information is needed by the general rules.

The following sections describe this categorization for each kind of construct:

- Sequence constructors: see [19.8.3 Classifying Sequence Constructors](#)
- Instructions: see [19.8.4 Classifying Instructions](#)
- Stylesheet functions: see [19.8.5 Classifying Stylesheet Functions](#)
- Attribute sets: see [19.8.6 Classifying Attribute Sets](#)
- Value templates: see [19.8.7 Classifying Value Templates](#)
- Expressions: see [19.8.8 Classifying Expressions](#)
- Patterns: see [19.8.10 Classifying Patterns](#)
- Calls to built-in functions: see [19.8.9 Classifying Calls to Built-In Functions](#)

19.8.1 General Rules for Streamability

[**DEFINITION:** Many `constructs` share the same streamability rules. These rules, referred to as the **general streamability rules**, are defined here.]

Examples of constructs that use these rules are: an arithmetic expression, an `attribute value template`, a `sequence constructor`, the `xsl:value-of` instruction, and a call to the `doc`^{FO30} function.

The rules determine both the `posture` and `sweep` of a construct. To determine the posture and sweep of a construct C , assuming these general rules are applicable to that kind of construct:

1. For each `operand` of C :
 - a. Establish:
 - i. The `static type` T of the operand (see [19.1 Determining the Static Type of a Construct](#)).

Note:

The static type is a [U-type](#). For example, the static type of the expression `(@*, *)` is $U\{element(), attribute()\}$.

- ii. The [sweep](#) S and [posture](#) P of the operand (by applying the rules in this section [19.8 Classifying Constructs](#) to that operand, recursively).
 - iii. The [operand usage](#) U corresponding to the [role](#) of the operand within C (from the information in this section [19.8 Classifying Constructs](#)).
- b. Compute the adjusted sweep S' of the [operand](#) by taking the first of the following that applies:
- i. If S is [free-ranging](#) or P is [roaming](#), then S' is [free-ranging](#). (In this case the posture and sweep of C are [roaming](#) and [free-ranging](#), regardless of any other operands.)
 - ii. If P is [grounded](#), then S' is S .
 - iii. Otherwise (P is not [grounded](#), which implies that the [operand](#) is capable of returning streamed nodes), compute S' as follows:
 - A. Compute the adjusted usage U' as follows:
 - I. If U is [absorption](#) and the intersection of T with $U\{element(), document-node()\}$ is $U\{\}$ (that is, if T is a type that does not allow nodes with children), then U' is [inspection](#).

Note:

This is because the entire subtree of nodes such as text nodes is available without reading further data from the input stream.

II. Otherwise, U' is U .

- B. Compute the adjusted [sweep](#) S' from the table below:

Computing the Adjusted Sweep of an Expression

Posture (P)	Adjusted Usage (U')			
	Absorption	Inspection	Transmission	Navigation
Climbing	Free-ranging	S	S	Free-ranging
Striding	Consuming	S	S	Free-ranging
Crawling	Consuming	S	S	Free-ranging

- c. [DEFINITION: An [operand](#) is **potentially consuming** if at least one of the following conditions applies:

- i. The operand's adjusted [sweep](#) S' is [consuming](#).
- ii. The [operand usage](#) is [transmission](#) and the operand is not [grounded](#).

]

2. Having computed the adjusted sweep $S'(o)$ of each [operand](#) o , the [posture](#) and [sweep](#) of C are the first of the following that applies:
- a. If C has no operands, then [grounded](#) and [motionless](#).

- b. If any operand o has an adjusted sweep $S'(o)$ of free-ranging, then roaming and free-ranging.
- c. If more than one operand is potentially consuming, then:
 - i. If all these operands form part of a choice operand group, then the posture of C is the combined posture of the operands in this group, and the sweep of C is the widest sweep of the operands in this group
 - ii. If all these operands have $S' = \text{motionless}$, (which necessarily means they have $U' = U = \text{transmission}$) and if they all have the same posture P_0 , then motionless with posture P_0 .

Note:

For example, the expression $(@a, @b)$ is motionless and striding.

- iii. Otherwise, roaming and free-ranging.
- d. If exactly one operand o is potentially consuming, then:
 - i. If o is a higher-order operand of C , then roaming and free-ranging.
 - ii. If the operand usage of o is absorption or inspection, then grounded and consuming.
 - iii. If the posture of o is crawling and C is a function call of a built-in function whose signature indicates a return type with a maximum cardinality of one then striding and the adjusted sweep of o .
- e. Otherwise (the operand usage of o is transmission), the posture and adjusted sweep of o .
- f. Otherwise (all operands are motionless) grounded and motionless.

Note:

Although this rule is written in general terms, the only functions that it applies to (at the time of publication) are head^{FO30}, exactly-one^{FO30}, and zero-or-one^{FO30}. This rule only applies if the argument usage is transmission (other cases having been handled by earlier rules); of the built-in functions, the three functions listed are the only ones having an argument with usage transmission and a return type with maximum cardinality one.

Note:

The rules ensure that if more than one [operand](#) is [consuming](#), that is, if more than one operand reads the subtree of the context node in a way that would cause the current position of the input stream to change, then the construct is not streamable.

The rules also prevent multiple streamed nodes being returned in the result of an expression if they are delivered by different operands. For example, the expression `count((., *))` is not guaranteed streamable. This is to make static analysis possible: the posture needs to be statically determined to ensure that streaming does not fail at execution time. It is permitted, however, for streamed nodes to be mixed in a sequence with non-streamed nodes or with atomic values; in this case the posture of the result will be that of the streamed nodes. It is also permitted to have multiple operands delivering streamed nodes in different branches of a conditional, provided the sweep and posture are compatible: for example `if (X) then @name else name` is guaranteed streamable.

Expressions that have more than one operand with usage [transmission](#), for example `(A, B)`, or `(A | B)`, or `insert-before(A, n, B)`, generally allow only one of these operands to select streamed nodes. The result of the expression will contain a mixture of streamed and grounded nodes, but its posture and sweep will be that of the streamed operand. The nodes in the result will not necessarily be in document order, but the subset of the nodes that are streamed will always be in document order.

19.8.2 Examples of the General Streamability Rules

This section provides some examples of how the general streamability rules operate. In each example, the emphasis is on the outermost construct shown; explanations for how the sweep and posture of its operands are derived are not given, though in many cases they are explained in earlier examples.

The examples assume that the context item type for evaluation of the expression shown is an element node, and that its posture is striding.

- `2 + 2` is grounded and motionless, because both the operands are grounded and motionless.
- `price * 2` is grounded and consuming, because one of the operands is consuming and the relevant operand usage is absorption.
- `price - discount` is roaming and free-ranging, because both the operands are consuming (and they are not members of a parallel operand group).
- `price * @discount` is grounded and consuming. The left-hand operand is consuming and the corresponding operand usage is absorption, while the right-hand operand is motionless, again with an operand usage of absorption, and its item type is `attribute()` which changes the effective usage to inspection.
- `a/b/c` is striding and consuming. This is determined not by the general streamability rules, but by the rules for path expressions in [19.8.8.8 Streamability of Path Expressions](#).
- `a//c` is crawling and consuming. This is similarly determined by the rules for path expressions in [19.8.8.8 Streamability of Path Expressions](#).
- `count(a/b/c)` is grounded and consuming, because the operand (the argument to the count function) is striding and consuming (see earlier example) and the operand usage is inspection.

- `sum(a/b/c)` is grounded and consuming, because the operand (the argument to the `sum` function) is striding and consuming (see earlier example) and the operand usage is absorption.
- `count(descendant::c)` is grounded and consuming, because the operand (the argument to the `count` function) is crawling and consuming (see earlier example) and the operand usage is inspection.
- `tail(descendant::c)` is crawling and consuming. The operand is crawling, the operand usage is transmission, so the posture and sweep of the result are the same as the posture and sweep of the consuming operand.
- `unordered(a|b)` is crawling and consuming. The operand (the argument to the `unordered` function) is crawling (see [19.8.8.4 Streamability of union, intersect, and except Expressions](#)), and the operand usage is transmission, so the posture and sweep of the result are the same as the posture and sweep of the consuming operand.
- `zero-or-one(descendant::c)` is striding and consuming. Although the operand is crawling, the operand usage is transmission and the cardinality of the expression is zero or one, so the posture of the result is striding. The same analysis applies to `exactly-one(descendant::c)` and to `head(descendant::c)`.
- `sum(descendant::c)` is grounded and consuming, because the operand (the argument to the `sum` function) is crawling and consuming (see earlier example) and the operand usage is absorption. In theory (although it is unlikely in practice) the selected `c` elements might be nested one inside another. The processor is expected to handle this situation, which may require some buffering. For example, given the untyped source document `<a><c><c>1</c><c>2</c><c>3</c></c>`, the result of the expression is 129 ($123 + 1 + 2 + 3$), and to evaluate this, a streaming processor will typically maintain a stack of buffers to accumulate the typed values of each of the four `c` elements during a single pass of the source document.
- `"Q{" || namespace-uri(.) || "}" || local-name(.)` is grounded and motionless. The two literal operands are grounded and motionless because they have no operands; the two function calls are grounded and motionless because they have a single operand that is striding and motionless, with an operand usage of inspection.
- `copy-of(./head/following-sibling::*)` is grounded and consuming. The left-hand operand `copy-of(./head)` is grounded and consuming because, under the rules in [19.8.8.8 Streamability of Path Expressions](#), its left-hand operand `copy-of(./)` is grounded and consuming. This in turn is because `.` is striding and motionless, and the operand usage is absorption.
- `if ($discounted) then price else discounted-price` is striding and consuming, because the two branches of the conditional are both striding and consuming, and they form a [choice operand group](#) with usage transmission.
- `if ($gratis) then 0 else price` is striding and consuming because there is only one consuming operand (the fact that it is part of a [choice operand group](#) does not affect the reasoning).
- `count((author, editor))` is roaming and free-ranging. The first argument to the `count` function is an expression with two operands, both having usage=transmission, and neither being grounded.
- `count((author | editor))` is grounded and consuming. A union expression is not subject to the general streamability rules; it has its own rules, defined in [19.8.8.4 Streamability of union, intersect, and except Expressions](#), which establish in this case that the argument to the `count`^{FO30} is [crawling](#) and [consuming](#). The `count`^{FO30} function does follow the general streamability rules, with an operand usage of [inspection](#): under rule 1(b)(iii)(B) the adjusted sweep is [consuming](#), and rule 2(d)(iii) then applies.
- `('{' , author, '}')` is striding and consuming. Exactly one operand is consuming; it has usage [transmission](#), so the result has the posture and sweep of that operand. (The formal analysis treats comma as a

binary operator, but the same result can be obtained by treating the content of the parenthesized expression as an expression with three operands.)

19.8.3 Classifying Sequence Constructors

The posture and sweep of a sequence constructor are determined by the general streamability rules.

The operand roles and their usages are:

1. The immediately contained instructions and literal result elements, including any xsl:on-empty or xsl:on-non-empty instructions. The operand usage for these operands is transmission.
2. Any text value templates appearing in text nodes within the sequence constructor, if text value templates are enabled. The operand usage for these operands is absorption.

Note:

Some consequences of these rules are:

1. An empty sequence constructor is motionless, and its posture is grounded.
2. A sequence constructor containing a single instruction has the same sweep and posture as that instruction. (This means that sequence constructors containing a single instruction can usefully be dropped from the construct tree.)
3. Informally, a sequence constructor is not streamable if it contains more than one instruction that moves the position of the input stream.
4. xsl:on-empty or xsl:on-non-empty instructions are not treated specially. For example, there is no attempt to take into account that they are mutually exclusive: if one is evaluated, the other will not be evaluated. In most use cases for these instructions, they will be motionless, so the additional complexity of doing more advanced analysis would rarely be justified.

19.8.4 Classifying Instructions

This section describes how instructions are classified with respect to their streamability. The criteria are given first for literal result elements and extension instructions, then for each XSLT instruction, listed alphabetically.

19.8.4.1 Streamability of Literal Result Elements

The posture and sweep of a literal result element follow the general streamability rules. The operand roles and their usages are:

1. The contained sequence constructor (usage absorption)
2. Any expressions contained in attribute value templates among the literal result element's attributes (usage absorption)
3. Any attribute sets named in the xsl:use-attribute-sets attribute (usage irrelevant, but can be taken as inspection).

Note:

In practice, a reference to an attribute set that is [declared-streamable](#) does not affect the analysis, while a reference to any other attribute set makes the literal result element [roaming](#) and [free-ranging](#).

19.8.4.2 Streamability of extension instructions

For a processor that recognizes an [extension instruction](#), the [posture](#) and [sweep](#) of the instruction are [implementation-defined](#).

For a processor that does not recognize an [extension instruction](#), the [posture](#) and [sweep](#) of the instruction are determined by applying the [general streamability rules](#). The [operand roles](#) and their [usages](#) are:

1. The [sequence constructors](#) contained in any [`xsl:fallback`](#) children (usage [transmission](#))

Instructions in the XSLT namespace that are present under the provisions for [forwards compatible behavior](#) are treated in the same way as unrecognized extension instructions.

Note:

These rules mean that if there is no [`xsl:fallback`](#) child instruction, the containing construct will be classified as streamable. However, any attempt to execute the instruction will lead to a dynamic error, so in fact, neither streamed nor unstreamed evaluation is possible.

19.8.4.3 Streamability of `xsl:analyze-string`

The [posture](#) and [sweep](#) of [`xsl:analyze-string`](#) follow the [general streamability rules](#). The [operand roles](#) and their [usages](#) are:

1. the [select expression](#) (usage [absorption](#));
2. the [regex attribute value template](#) (usage [absorption](#));
3. the sequence constructors contained in the [`xsl:matching-substring`](#) and [`xsl:non-matching-substring`](#) elements. These have usage [navigation](#), because they can be evaluated more than once. The [context posture](#) for the two sequence constructors is [grounded](#), reflecting the fact that their context item type is [xs:string](#).

Note:

In practice, the [sweep](#) of the instruction will usually be the same as the sweep of the [select expression](#), and its [posture](#) will be [grounded](#). Exceptions occur for example if the [regex attribute](#) is not [motionless](#), or if the contained sequence constructors refer to a grouping variable bound in a contained [`xsl:for-each-group`](#) instruction.

19.8.4.4 Streamability of `xsl:apply-imports`

The rules in this section apply also to [xsl:next-match](#).

The [posture](#) and [sweep](#) of these two instructions follow the [general streamability rules](#). The [operand roles](#) and their [usages](#) are:

1. An implicit operand: a context item expression (.), with usage [absorption](#);
2. The `select` attribute or contained [sequence constructor](#) of each `xsl:with-param` child element, with [type-determined usage](#) based on the type declared in the `xsl:with-param/@as` attribute, or `item()*` if absent.

Note:

The instruction will normally be [grounded](#) and [consuming](#), provided that nodes in a streamed document are not passed as parameters to the called template rule.

19.8.4.5 Streamability of `xsl:apply-templates`

If there is no `select` attribute, the following analysis assumes the presence of an implicit operand `select="child::node()"`.

The [posture](#) and [sweep](#) of the `xsl:apply-templates` instruction are the first of the following that apply:

1. If the `select` expression is [grounded](#), then the [posture](#) and [sweep](#) of the `xsl:apply-templates` instruction follow the [general streamability rules](#), with the [operand roles](#) and their [usages](#) as follows:
 - a. The `select` expression (the [operand usage](#)) is irrelevant, but can be taken as [absorption](#)
 - b. The `select` expressions and contained sequence constructors of any child `xsl:with-param` elements (usage [type-determined](#), based on the type in the `xsl:with-param/@as` attribute, defaulting to `item()*`)
 - c. Any attribute value templates appearing in attributes of a child `xsl:sort` instruction (usage [absorption](#))
 - d. The `select` expression or contained sequence constructor of any `xsl:sort` children, assessed with a [context posture](#) of [grounded](#) (usage [absorption](#)).

For example, `<xsl:apply-templates select="copy-of(.)"/>` is [grounded](#) and [consuming](#).

2. If there is an `xsl:sort` child element, then [roaming](#) and [free-ranging](#).
3. If the implicit or explicit `mode` attribute identifies a [mode](#) that is not declared with `streamable="yes"`, then [roaming](#) and [free-ranging](#).

Note:

When `mode="#current"` is specified, this is treated as equivalent to specifying a streamable mode; although it is not known statically what the mode will be, it is always the case that if the template is invoked with a streamed node as the context item, then the current mode must be a streamable mode.

4. If the `select` expression is [climbing](#) or [crawling](#), then [roaming](#) and [free-ranging](#)
5. Otherwise, the [posture](#) and [sweep](#) of the `xsl:apply-templates` instruction follow the [general streamability rules](#). The [operand roles](#) and their [usages](#) are as follows:
 - a. The (explicit or implicit) `select` expression, with usage [absorption](#);

- b. The `select` attribute or contained [sequence constructor](#) of each `xsl:with-param` child element, with [type-determined usage](#) based on the type declared in the `xsl:with-param/@as` attribute, or `item()*` if absent.

19.8.4.6 [Streamability of `xsl:assert`](#)

The [posture](#) and [sweep](#) of [`xsl:assert`](#) follow the [general streamability rules](#). The [operand roles](#) and their [usages](#) are as follows:

1. The `test` expression (usage [inspection](#))
2. The `select` expression (usage [absorption](#))
3. The `error-code` attribute value template (usage [absorption](#))
4. The contained [sequence constructor](#) (usage [absorption](#)).

19.8.4.7 [Streamability of `xsl:attribute`](#)

The [posture](#) and [sweep](#) of [`xsl:attribute`](#) follow the [general streamability rules](#). The [operand roles](#) and their [usages](#) are as follows:

1. The `name` attribute value template (usage [absorption](#))
2. The `namespace` attribute value template (usage [absorption](#))
3. The `select` expression (usage [absorption](#))
4. The `separator` attribute value template (usage [absorption](#))
5. The contained [sequence constructor](#) (usage [absorption](#)).

19.8.4.8 [Streamability of `xsl:break`](#)

The [posture](#) and [sweep](#) of [`xsl:break`](#) follow the [general streamability rules](#). The [operand roles](#) and their [usages](#) are as follows:

1. The `select` expression (usage [transmission](#))
2. The contained [sequence constructor](#) (usage [transmission](#)).

19.8.4.9 [Streamability of `xsl:call-template`](#)

The [posture](#) and [sweep](#) of [`xsl:call-template`](#) follow the [general streamability rules](#). The [operand roles](#) and their [usages](#) are as follows:

1. Unless the referenced template has a child `xsl:context-item` element with the attribute `use="prohibited"`, there is an implicit operand, a context item expression `(.)`: its [operand usage](#) is the [type-determined usage](#) based on the type declared in the `xsl:context-item/@as` attribute of the target named template, defaulting to `item()*` if absent.

2. The `select` expression or sequence constructor content of any contained `xsl:with-param` child element: its operand usage is the type-determined usage based on the type declared in the `xsl:with-param/@as` attribute, or the `xsl:param/@as` attribute of the corresponding parameter on the target named template, whichever is more restrictive, defaulting to `item()*` if both are absent.

Note:

Calling `xsl:call-template` will usually make stylesheet code unstreamable if a streamed node is passed explicitly or implicitly to the called template, unless it is atomized by declaring the expected type to be atomic.

19.8.4.10 Streamability of `xsl:choose`

The posture and sweep of `xsl:choose` follow the general streamability rules. The operand roles and their usages are as follows:

1. The `test` attribute of contained `xsl:when` elements (usage inspection).
2. The sequence constructors contained within `xsl:when` and `xsl:otherwise` child elements (usage transmission). These sequence constructor operands form a choice operand group.

Note:

The effect is to allow either of the following:

1. Any or all of the sequence constructors in `xsl:when` and `xsl:otherwise` branch may be consuming, in which case the `test` expressions must all be motionless.
2. Any one of the `test` expressions may be consuming, in which case all the other `test` expressions, and all the sequence constructors, must be motionless.

19.8.4.11 Streamability of `xsl:comment`

The posture and sweep of `xsl:comment` follow the general streamability rules. The operand roles and their usages are as follows:

1. The `select` expression (usage absorption)
2. The contained sequence constructor (usage absorption).

19.8.4.12 Streamability of `xsl:copy`

The posture and sweep of `xsl:copy` follow the general streamability rules. The operand roles and their usages are as follows:

1. The expression in the `select` attribute, defaulting to a context item expression `(.)` (usage inspection)

2. The contained sequence constructor (usage [absorption](#)), assessed with [context posture](#) and context item type based on the `select` expression if present, or the outer focus otherwise.
3. Any [attribute sets](#) named in the `use-attribute-sets` attribute (usage irrelevant, but can be taken as [inspection](#)).

Note:

In practice, a reference to an attribute set that is [declared-streamable](#) does not affect the analysis, while a reference to any other attribute set makes the [`xsl:copy`](#) instruction [roaming](#) and [free-ranging](#).

Note:

The effect of these rules is that when a `select` attribute is present, the sequence constructor contained by the [`xsl:copy`](#) instruction is deemed to be a [higher-order operand](#) of the instruction, even though it can only be evaluated once.

This has the practical consequence that the following example is not [guaranteed-streamable](#), even though it is possible to imagine a strategy for streamed evaluation:

```
<xsl:for-each-group select="product" group-adjacent="@category">
    <xsl:copy select="..">
        <xsl:copy-of select="current-group()" />
    </xsl:copy>
</xsl:for-each-group>
```

A workaround in this case might be to rewrite the code as follows:

```
<xsl:for-each-group select="product" group-adjacent="@category">
    <xsl:element name="{name(..)}" namespace-uri="{namespace-uri(..)}">
        <xsl:copy-of select="current-group()" />
    </xsl:element>
</xsl:for-each-group>
```

19.8.4.13 Streamability of `xsl:copy-of`

The [posture](#) and [sweep](#) of [`xsl:copy-of`](#) follow the [general streamability rules](#). The [operand roles](#) and their [usages](#) are as follows:

1. The `select` expression (usage [absorption](#)).

19.8.4.14 Streamability of `xsl:document`

The [posture](#) and [sweep](#) of [`xsl:document`](#) follow the [general streamability rules](#). The [operand roles](#) and their [usages](#) are as follows:

1. The contained [sequence constructor](#) (usage [absorption](#)).

19.8.4.15 Streamability of `xsl:element`

The posture and sweep of `xsl:element` follow the general streamability rules. The operand roles and their usages are as follows:

1. The `name` attribute value template (usage [absorption](#))
2. The `namespace` attribute value template (usage [absorption](#))
3. Any attribute sets named in the `use-attribute-sets` attribute (usage irrelevant, but can be taken as [inspection](#)).

Note:

In practice, a reference to an attribute set that is [declared-streamable](#) does not affect the analysis, while a reference to any other attribute set makes the `xsl:element` instruction [roaming](#) and [free-ranging](#).

4. The contained sequence constructor (usage [absorption](#)).

19.8.4.16 Streamability of `xsl:evaluate`

The posture and sweep of `xsl:evaluate` follow the general streamability rules. The operand roles and their usages are as follows:

1. The `xpath` expression (usage [absorption](#))
2. The `context-item` expression (usage [navigation](#))
3. The `with-params` expression (usage [navigation](#))
4. The `base-uri` attribute value template (usage [absorption](#))
5. The `namespace-context` expression (usage [inspection](#))
6. The `schema-aware` attribute value template (usage [absorption](#))
7. The `select` attributes and contained sequence constructors of any `xsl:with-param` child elements (usage [type-determined](#), based on the type in the `xsl:with-param/@as` attribute, defaulting to `item()*`)

Note:

In practice, code containing an `xsl:evaluate` instruction will usually be streamable provided that streamed nodes are not passed to the dynamic expression either as the context item or as the value of a parameter.

19.8.4.17 Streamability of `xsl:fallback`

The posture and sweep of the `xsl:fallback` instruction depend on whether the processor is performing fallback (which is known statically).

If the processor is performing fallback, then the [posture](#) and [sweep](#) of the [`xsl:fallback`](#) instruction are the posture and sweep of the contained sequence constructor.

If the processor is not performing fallback, then the instruction is [grounded](#) and [motionless](#).

19.8.4.18 [*Streamability of `xsl:for-each`*](#)

The [posture](#) and [sweep](#) of the [`xsl:for-each`](#) instruction are the first of the following that applies:

1. If the [select](#) expression is [grounded](#), then the [posture](#) and [sweep](#) of the [`xsl:for-each`](#) instruction follow the [general streamability rules](#), with the [operand roles](#) and their [usages](#) as follows:
 - a. The [select](#) expression (the [operand usage](#) is irrelevant, but can be taken as [inspection](#))
 - b. The contained [sequence constructor](#) (usage [transmission](#)). This is a [higher-order operand](#); its context posture is [grounded](#).
 - c. Any attribute value templates appearing in attributes of a child [`xsl:sort`](#) instruction (usage [absorption](#))
 - d. The [select](#) expression or contained sequence constructor of any [`xsl:sort`](#) children, assessed with a [context posture](#) of [grounded](#) (usage [absorption](#)). These are [higher-order operands](#); their context posture is [grounded](#).
2. If there is an [`xsl:sort`](#) child element, then [roaming](#) and [free-ranging](#).
3. If the [posture](#) of the [select](#) expression is [crawling](#) and the [sweep](#) of the contained [sequence constructor](#) is [consuming](#), then [roaming](#) and [free-ranging](#).
4. Otherwise:
 - a. The [posture](#) of the instruction is the [posture](#) of the contained [sequence constructor](#), assessed with the [context posture](#) and context item type set to the [posture](#) and type of the [select](#) expression.
 - b. The [sweep](#) of the instruction is the wider of the [sweep](#) of the [select](#) expression and the [sweep](#) of the contained [sequence constructor](#).

Note:

The ordering of sweep values is in increasing order: [motionless](#), [consuming](#), [free-ranging](#).

Note:

Because the body of the [`xsl:for-each`](#) instruction is a [higher-order operand](#) of the instruction, any variable reference within the body that is bound to a [streaming parameter](#) of a containing [stylesheet function](#) will not be singular, which in many cases will make the entire function non-streamable.

19.8.4.19 [*Streamability of `xsl:for-each-group`*](#)

The [posture](#) and [sweep](#) of the [`xsl:for-each-group`](#) instruction are the first of the following that applies:

1. If the [select](#) expression is [grounded](#), then the [posture](#) and [sweep](#) of the [`xsl:for-each-group`](#) instruction follow the [general streamability rules](#), with the [operand roles](#) and their [usages](#) as follows:
 - a. The [select](#) expression (the [operand usage](#) is irrelevant, but can be taken as [inspection](#))

- b. The `collation` attribute value template (usage [absorption](#))
 - c. Any attribute value templates appearing in attributes of a child `xsl:sort` instruction (usage [absorption](#))
 - d. The group-by or group-adjacent expression, assessed with a [context posture](#) of [grounded](#) (usage [absorption](#)).
 - e. The `select` expression or contained sequence constructor of any `xsl:sort` children, assessed with a [context posture](#) of [grounded](#) (usage [absorption](#)).
 - f. The group-starting-with or group-ending-with patterns if present; these are [higher-order operands](#) with usage [inspection](#).
2. If there is a group-by attribute and the instruction is not a child of `xsl:fork`, then [roaming](#) and [free-ranging](#).
 3. If there is a group-by or group-adjacent attribute that is not [motionless](#), then [roaming](#) and [free-ranging](#).
 4. If there is an `xsl:sort` child element and the instruction is not a child of `xsl:fork`, then [roaming](#) and [free-ranging](#).
 5. If the [posture](#) of the `select` expression is [crawling](#) and the [sweep](#) of the contained [sequence constructor](#) is [consuming](#), then [roaming](#) and [free-ranging](#).
 6. Otherwise:
 - a. The [posture](#) of the instruction is the [posture](#) of the contained [sequence constructor](#), assessed with the [context posture](#) and context item type set to the [posture](#) and type of the `select` expression.
 - b. The [sweep](#) of the instruction is the wider of the [sweeps](#) of the `select` expression and the contained [sequence constructor](#), where the ordering of increasing width is [motionless](#), [consuming](#), [free-ranging](#).

Note:

Because the body of the `xsl:for-each-group` instruction is a [higher-order operand](#) of the instruction, any variable reference within the body that is bound to a [streaming parameter](#) of a containing [stylesheet function](#) will not be singular, which in many cases will make the entire function non-streamable.

Note:

The above rules do not explicitly mention any constraints on the presence or absence of a call on the [current-group](#) function. In practice, however, this plays an important role. In the most common case, the `select` expression of `xsl:for-each-group` is likely to be striding, for example an expression such as `select="*"`. Any call on [current-group](#) associated with this `xsl:for-each-group` instruction will ordinarily be [striding](#) and [consuming](#), which is consistent with streaming provided there is only one such call, and if it appears in a suitable context (for example, not within a predicate). If there is more than one call, or if it appears in an unsuitable context (for example, within a predicate), then this will have the same effect as multiple appearances of other consuming expressions: the construct as a whole will be free-ranging. These rules are not spelled out explicitly, but rather emerge as a consequence of the general streamability rules.

19.8.4.20 Streamability of `xsl:fork`

The [posture](#) and [sweep](#) of `xsl:fork` are the first of the following that applies:

1. If there is a child `xsl:for-each-group` instruction, then the `posture` and the `sweep` of that instruction.
2. If there are no child `xsl:sequence` instructions (other than `xsl:fallback`), then `grounded` and `motionless`.
3. If there is a child `xsl:sequence` instruction whose `posture` is not `grounded`, then `roaming` and `free-ranging`.
4. Otherwise, the `posture` is `grounded`, and the `sweep` is the widest sweep of the `xsl:sequence` child instructions.

Note:

None of the branches of `xsl:fork` can return streamed nodes. The reason for this is that `xsl:fork` has to assemble its results in the correct order, and streamed nodes cannot be re-ordered.

The effect of the rules is that each of the child `xsl:sequence` instructions can independently consume the streamed input document, provided that the result of each child instruction is `grounded`.

Thus the following example is streamable:

```
<xsl:fork>
  <xsl:sequence select="copy-of(author)" />
  <xsl:sequence select="copy-of(editor)" />
</xsl:fork>
```

While the following is not streamable, because it returns streamed nodes in an order that might not be document order:

```
<xsl:fork>
  <xsl:sequence select="author" />
  <xsl:sequence select="editor" />
</xsl:fork>
```

19.8.4.21 *Streamability of xsl:if*

The `posture` and `sweep` of `xsl:if` follow the `general streamability rules`. The `operand roles` and their `usages` are as follows:

1. The `test` expression (usage `inspection`)
2. The contained `sequence constructor` (usage `transmission`).

19.8.4.22 *Streamability of xsl:iterate*

The `posture` and `sweep` of the `xsl:iterate` instruction are the first of the following that applies:

1. If the `select` expression is `grounded`, then the `posture` and `sweep` of the `xsl:iterate` instruction follow the `general streamability rules`, with the `operand roles` and their `usages` as follows:
 - a. The `select` expression (the `operand usage` is irrelevant, but can be taken as `inspection`)
 - b. The `select` expression or contained sequence constructor of any `xsl:param` children (usage `navigation`)

- c. The sequence constructor contained within the `xsl:iterate` instruction itself, assessed with its context item type and `context posture` based on the `select` expression (usage [transmission](#))
- d. The `select` expression or contained sequence constructor of any child `xsl:on-completion` element, assessed with a context item type of `xs:error` and a `context posture` of `roaming` to reflect the fact that any attempt to reference the context item within the `xsl:on-completion` element is an error (usage [transmission](#))

Note:

The `on-completion` element can cause the instruction to become non-streamable if, for example, it contains a call on `current-group` or a variable reference bound to a [streaming parameter](#).

2. If there is an `xsl:param` child whose initializing `select` expression or `sequence constructor` is not `grounded` and `motionless`, then `roaming` and `free-ranging`.
3. If there is an `xsl:on-completion` child whose `select` expression or `sequence constructor` is not `grounded` and `motionless`, then `roaming` and `free-ranging`.
4. If the `posture` of the `select` expression is `crawling` and the `sweep` of the contained `sequence constructor` is `consuming`, then `roaming` and `free-ranging`.
5. Otherwise:
 - a. The `posture` of the instruction is the `posture` of the contained `sequence constructor`, assessed with the `context posture` and context item type set to the `posture` and type of the `select` expression.
 - b. The `sweep` of the instruction is the wider of the `sweeps` of the `select` expression and the contained `sequence constructor`, where the ordering of increasing width is `motionless`, `consuming`, `free-ranging`.

Note:

If any `xsl:break` or `xsl:next-iteration` instructions appear within the sequence constructor, their `posture` and `sweep` will be assessed in the course of evaluating the `posture` and `sweep` of the sequence constructor, by reference to the rules in [19.8.4.8 Streamability of xsl:break](#) and [19.8.4.28 Streamability of xsl:next-iteration](#) respectively.

Note:

Because the body of the `xsl:iterate` instruction is a `higher-order operand` of the instruction, any variable reference within the body that is bound to a `streaming parameter` of a containing `stylesheet function` will not be singular, which in many cases will make the entire function non-streamable.

19.8.4.23 [Streamability of xsl:map](#)

The `posture` and `sweep` of the `xsl:map` instruction are determined by the first of the following that applies:

1. If the sequence constructor within the instruction consists exclusively of `xsl:map-entry` instructions (and `xsl:fallback` instructions, which are ignored), then:

- a. If any of these `xsl:map-entry` children is `roaming` or `free-ranging`, then `roaming` and `free-ranging`;
 - b. Otherwise, `grounded` and the widest sweep of the `xsl:map-entry` children.
2. Otherwise, the `posture` and `sweep` of the `xsl:map` instruction are the posture and sweep of the contained `sequence constructor`.

Note:

See discussion in [21.6 Maps and Streaming](#).

The effect of the rules is that it is possible to compute multiple map entries in a single pass of the streamed input document. For example, the following is streamable:

```
<xsl:map>
  <xsl:map-entry key="'authors'" select="copy-of(author)"/>
  <xsl:map-entry key="'editors'" select="copy-of(editor)"/>
</xsl:map>
```

The call on `copy-of` is necessary to ensure that the content of the map entry is grounded; it is not possible to create a map whose entries contain references to streamed nodes.

19.8.4.24 Streamability of `xsl:map-entry`

The `posture` and `sweep` of `xsl:map-entry` follow the [general streamability rules](#). The [operand roles](#) and their [usages](#) are as follows:

1. The `key` expression (usage [absorption](#))
2. The `select` expression (usage [navigation](#))

Note:

This effectively means that the `select` expression must not return nodes from a streamed input document.

3. The contained `sequence constructor` (usage [navigation](#)).

19.8.4.25 Streamability of `xsl:merge`

Note:

This section is concerned with the (not very interesting) impact of the `xsl:merge` instruction on the streamability of its containing template rule or `xsl:source-document` instruction.

For the (more important) rules concerning the way in which `xsl:merge` performs streamed processing of its own inputs, see [15.4 Streamable Merging](#).

The `posture` and `sweep` of `xsl:merge` are as follows:

1. If every `xsl:merge-source` child element satisfies all the following conditions:
 - a. The expression in the `for-each-item` attribute is either absent, or grounded and motionless;
 - b. The expression in the `for-each-source` attribute is either absent, or grounded and motionless;
 - c. Either at least one of the attributes `for-each-item` and `for-each-source` is present, or the expression in the `select` attribute is grounded and motionless
- then the `xsl:merge` instruction is grounded and motionless.
2. Otherwise, the `xsl:merge` instruction roaming and free-ranging.

[19.8.4.26 Streamability of `xsl:message`](#)

The posture and sweep of `xsl:message` follow the general streamability rules. The operand roles and their usages are as follows:

1. The `select` expression (usage absorption)
2. The `terminate` attribute value template (usage absorption)
3. The `error-code` attribute value template (usage absorption)
4. The contained sequence constructor (usage absorption).

[19.8.4.27 Streamability of `xsl:namespace`](#)

The posture and sweep of `xsl:namespace` follow the general streamability rules. The operand roles and their usages are as follows:

1. The `name` attribute value template (usage absorption)
2. The `select` expression (usage absorption)
3. The contained sequence constructor (usage absorption).

[19.8.4.28 Streamability of `xsl:next-iteration`](#)

The posture and sweep of `xsl:next-iteration` follow the general streamability rules. The operand roles and their usages are as follows:

1. The `select` expression or sequence constructor content of any contained `xsl:with-param` child element: its operand usage is the type-determined usage based on the type declared in the `xsl:with-param/@as` attribute, or the `xsl:param/@as` attribute of the corresponding parameter on the containing `xsl:iterate` instruction, whichever is more restrictive, defaulting to `item()*` if both are absent.

[19.8.4.29 Streamability of `xsl:next-match`](#)

The rules are the same as for `xsl:apply-imports`: see [19.8.4.4 Streamability of `xsl:apply-imports`](#).

19.8.4.30 Streamability of `xsl:number`

The `posture` and `sweep` of `xsl:number` follow the [general streamability rules](#). The `operand roles` and their `usages` are as follows:

1. The `value` attribute if present: usage [absorption](#)
2. The `select` attribute if there is no `value` attribute, defaulting to the context item expression `(.)` if the `select` attribute is also absent: usage [navigation](#)
3. The attribute value templates in the `format`, `lang`, `letter-value`, `ordinal`, `start-at`, `grouping-separator`, and `grouping-size` attributes (usage [absorption](#))
4. The `from` and `count` patterns if present. These can be treated as [higher-order operands](#) with usage [inspection](#), though neither of these properties affects the outcome.

Note:

The effect of these rules is that `xsl:number` can be used for formatting of numbers supplied directly using the `value` attribute, and also for numbering of nodes in a non-streamed document, but it cannot be used for numbering streamed nodes.

In practice the rules depend very little on the `from` and `count` patterns. This is because when the instruction is applied to a streamed node, the instruction will be [free-ranging](#) regardless of these patterns; while if it is applied to a grounded node or atomic value, the instruction will normally be [motionless](#) regardless of the values of these patterns. The pattern does matter, however, if it contains a variable reference bound to a [streaming parameter](#); because such a reference occurs within a [higher-order operand](#) of the `xsl:number` instruction, its presence automatically makes the variable reference [free-ranging](#), which in turn ensures that the containing stylesheet function is not [guaranteed-streamable](#).

19.8.4.31 Streamability of `xsl:on-empty`

The streamability rules for the `xsl:on-empty` instruction are the same as the rules for `xsl:sequence`: see [19.8.4.36 Streamability of xsl:sequence](#).

Note:

The streamability rules for a sequence constructor containing an `xsl:on-empty` instruction are given in [19.8.3 Classifying Sequence Constructors](#).

19.8.4.32 Streamability of `xsl:on-non-empty`

The streamability rules for the `xsl:on-non-empty` instruction are the same as the rules for `xsl:sequence`: see [19.8.4.36 Streamability of xsl:sequence](#).

Note:

The streamability rules for a sequence constructor containing an [xsl:on-non-empty](#) instruction are given in [19.8.3 Classifying Sequence Constructors](#).

19.8.4.33 Streamability of xsl:perform-sort

The [posture](#) and [sweep](#) of [xsl:perform-sort](#) follow the [general streamability rules](#). The [operand roles](#) and their [usages](#) are as follows:

1. The expression in the **select** attribute: usage [navigation](#) (because order is not preserved)
2. The expressions in the attribute value templates of [xsl:sort](#) child elements: usage [absorption](#)
3. The expression in the **select** attribute or contained sequence constructor in child [xsl:sort](#) child elements, with usage [absorption](#), assessed with [context posture](#) based on the expression in the [xsl:perform-sort/@select](#) attribute.

Note:

In practice, the [xsl:perform-sort](#) instruction cannot be used to sort nodes from the streamed input document, but it can be used to sort atomic values or [grounded](#) nodes, for example a copy of nodes from the streamed document made using the [copy-of](#) function.

19.8.4.34 Streamability of xsl:processing-instruction

The [posture](#) and [sweep](#) of [xsl:processing-instruction](#) follow the [general streamability rules](#). The [operand roles](#) and their [usages](#) are as follows:

1. The **name** attribute value template (usage [absorption](#))
2. The **select** expression (usage [absorption](#))
3. The contained [sequence constructor](#) (usage [absorption](#)).

19.8.4.35 Streamability of xsl:result-document

The [posture](#) and [sweep](#) of [xsl:result-document](#) follow the [general streamability rules](#). The [operand roles](#) and their [usages](#) are as follows:

1. The **href** attribute value template (usage [absorption](#))
2. The attribute value templates containing serialization properties (usage [absorption](#))
3. The contained [sequence constructor](#) (usage [absorption](#)).

19.8.4.36 Streamability of xsl:sequence

The [posture](#) and [sweep](#) of [`xsl:sequence`](#) follow the [general streamability rules](#). The [operand roles](#) and their [usages](#) are as follows:

1. The `select` attribute value template (usage [transmission](#))
2. The contained [sequence constructor](#) (usage [transmission](#)).

19.8.4.37 [*Streamability of `xsl:source-document`*](#)

Note:

The concern here is with the impact of [`xsl:source-document`](#) on any streaming template, or ancestor [`xsl:source-document`](#) instruction, and not with the streamed processing of the document accessed using the `xsl:source-document/@href` attribute.

The streamability of the document opened by the [`xsl:source-document`](#) instruction is not assessed using the rules in this section; it depends only on the streamability properties of the contained sequence constructor, as described in [18.1 The `xsl:source-document` Instruction](#)

The [posture](#) and [sweep](#) of [`xsl:source-document`](#) are the first of the following that applies:

1. If the contained sequence constructor contains, at any depth, a call on the [current-group](#) function whose nearest containing [`xsl:for-each-group`](#) instruction exists and is an ancestor of the [`xsl:source-document`](#) instruction, then [roaming](#) and [free-ranging](#).
2. If the contained sequence constructor contains, at any depth, a call on the [current-merge-group](#) function whose nearest containing [`xsl:merge`](#) instruction exists and is an ancestor of the [`xsl:source-document`](#) instruction, then [roaming](#) and [free-ranging](#).
3. Otherwise, the [posture](#) is [grounded](#) and the [sweep](#) is the [sweep](#) of the `href` attribute value template.

19.8.4.38 [*Streamability of `xsl:text`*](#)

The [posture](#) and [sweep](#) of [`xsl:text`](#) follow the [general streamability rules](#). There are no operands.

Note:

The instruction is therefore [grounded](#) and [motionless](#).

19.8.4.39 [*Streamability of `xsl:try`*](#)

The [posture](#) and [sweep](#) of the [`xsl:try`](#) instruction follow the [general streamability rules](#). The [operand roles](#) and [usages](#) are as follows:

1. The `select` expression or contained [sequence constructor](#) of the [`xsl:try`](#) element. This has [operand usage transmission](#). (Note that the [`xsl:catch`](#) children of [`xsl:try`](#) are not part of the sequence constructor and therefore not part of this operand.)

2. The `select` expressions and/or contained [sequence constructor](#) of the `xsl:catch` child elements. These form a [choice operand group](#) with [operand usage transmission](#).

Note:

The overall effect of these rules is that either the `xsl:try` branch or the `xsl:catch` branch may consume the streamed input, but not both. If there is more than one `xsl:catch` branch then they may all consume the input, since only one of these branches can be evaluated.

19.8.4.40 [Streamability of `xsl:value-of`](#)

The [posture](#) and [sweep](#) of `xsl:value-of` follow the [general streamability rules](#). The [operand roles](#) and their [usages](#) are as follows:

1. The `select` expression (usage [absorption](#))
2. The `separator` attribute value template (usage [absorption](#))
3. The contained [sequence constructor](#) (usage [absorption](#)).

19.8.4.41 [Streamability of `xsl:variable`](#)

The [posture](#) and [sweep](#) of `xsl:variable` follow the [general streamability rules](#). The [operand roles](#) and their [usages](#) depend on the `as` attribute, as follows:

1. If there is an `as` attribute, then:
 - a. The `select` expression (with [type-determined usage](#) based on the `as` attribute).
 - b. The contained [sequence constructor](#) (with [type-determined usage](#) based on the `as` attribute).
2. If there is no `as` attribute, then:
 - a. The `select` expression (usage [navigation](#)).
 - b. The contained [sequence constructor](#) (usage [absorption](#)).

Note:

The effect of the initialization expression having usage [navigation](#) is that it is not possible in streamable constructs to bind a variable to a node in a streamed document.

19.8.4.42 [Streamability of `xsl:where-populated`](#)

The [posture](#) and [sweep](#) of an `xsl:where-populated` instruction are the [posture](#) and [sweep](#) of the contained [sequence constructor](#).

19.8.5 [Classifying Stylesheet Functions](#)

Under specific conditions, described in this section, a stylesheet function can be used to process nodes from a [streamed input document](#).

[**DEFINITION:** Stylesheet functions belong to one of a number of **streamability categories**: the choice of category characterizes the way in which the function handles streamed input.]

The [category](#) to which a function belongs is declared in the [streamability](#) attribute of the [xsl:function](#) declaration, and defaults to [unclassified](#).

The streamability categories defined in this specification are: [unclassified](#), [absorbing](#), [inspection](#), [filter](#), [shallow-descent](#), [deep-descent](#), and [ascent](#). It is also possible to specify the streamability category as a QName in an [implementation-defined](#) namespace, in which case the streamability rules are [implementation-defined](#); a processor that does not recognize a category defined in this way **MUST** analyze the function as if [streamability="unclassified"](#) were specified.

A stylesheet function is [declared-streamable](#) if the [xsl:function](#) declaration has a [streamability](#) attribute with a value other than [unclassified](#).

The only [category](#) permitted for a zero-arity function (one with no arguments) is [unclassified](#). All function calls to zero-arity stylesheet functions are [grounded](#) and [motionless](#).

In general (subject to more detailed rules below), a node belonging to a [streamed document](#) can be present within the value of an argument of a call on a [stylesheet function](#) only if one of the following conditions is true:

1. The stylesheet function is [declared-streamable](#), and the argument in question is the first argument of the function call.
2. The corresponding [function parameter](#) is declared with a [required type](#) that triggers [atomization](#) of any supplied node.

[**DEFINITION:** The first [parameter](#) of a [declared-streamable stylesheet function](#) is referred to as a **streaming parameter**.]

Note:

If a stylesheet function returns streamed nodes, then these nodes can only derive from streamed nodes passed in an argument to the function. This is because streamed nodes cannot be bound to global variables, and they cannot be returned by an [xsl:source-document](#) instruction within the function body (the result of [xsl:source-document](#) is always grounded).

The choice of [category](#) places constraints on the function body, and also on calls to the function. These constraints are defined below, separately for each category. A function is [guaranteed-streamable](#) only if the constraints are satisfied, and a static function call is guaranteed-streamable only if the function is guaranteed-streamable and the function call itself satisfies the constraints for the chosen category.

Dynamic function calls are [guaranteed-streamable](#) only in trivial cases, for example where the function signature indicates that an argument is required to be a text node or an attribute node. For details, see [19.8.8.11 Streamability of Dynamic Function Calls](#).

The constraints on the function body are expressed in terms of the [posture](#) and [sweep](#) of the function result. The [posture](#) and [sweep](#) of the function result are the [type-adjusted posture and sweep](#) of the [sequence constructor](#)

contained within the [xsl:function](#) element, given the declared return type of the function, which defaults to `item()*`.

Note:

Determining the [posture](#) and [sweep](#) of the function result requires first determining the [posture](#) and [sweep](#) of the contained [sequence constructor](#), which is done according to the rules in [19.8.3 Classifying Sequence Constructors](#). This in turn will usually involve examination of variable references that are bound to the function's parameters. The analysis of these variable references is described in [19.8.8.12 Streamability of Variable References](#).

If the function is [declared-streamable](#) but does not satisfy the constraints that make it [guaranteed-streamable](#), the consequences are explained in [19.10 Streamability Guarantees](#).

If a stylesheet function is overridden in another package (using [xsl:override](#)), then the overriding stylesheet function must belong to the same [streamability category](#) as the function that it overrides. This ensures that overriding a function cannot affect the streamability of calls to that function.

The rules for each [streamability category](#) are given in the following sections.

19.8.5.1 *Streamability Category: unclassified*

Informal description: Functions in this category cannot be called with streamed nodes supplied in an argument, unless the function signature causes such nodes to be atomized.

Rules for the function signature: there are no constraints.

Rules for the function body: there are no constraints.

Rules for references to the streaming parameter: not applicable, because there is no streaming parameter.

Rules for function calls: the [general streamability rules](#) apply. The operands are the expressions appearing in the argument list of the function call, with the [operand usage](#) of each operand being the [type-determined usage](#) based on the declared type of the corresponding parameter in the function signature.

Example: An unclassified stylesheet function that accepts nodes

The [streamability category](#) is unclassified.

```
<xsl:function name="f:exclude-first" as="node()>*">
  <xsl:param name="nodes" as="node()*/>
  <xsl:sequence select="$nodes[not(node-name() = preceding-sibling::*/node-
  name())]"/>
</xsl:function>
```

The effect of the rules is that a call to this function is guaranteed streamable if and only if the sequence supplied as the value of the \$nodes argument is [grounded](#) (that is, it contains no streamed nodes).

Example: An unclassified stylesheet function that accepts atomic values

The [streamability category](#) is unclassified.

```
<xsl:function name="f:min" as="xs:integer">
  <xsl:param name="arg0" as="xs:integer"/>
  <xsl:param name="arg1" as="xs:integer"/>
  <xsl:sequence select="min(($arg0, $arg1))"/>
</xsl:function>
```

The effect of the rules is that a call to this function is streamable under similar circumstances to those that apply to a binary operator such as `+`. For example, a call is streamable if two atomic values are supplied, or if two attribute nodes are supplied, whether from streamed or unstreamed documents. The main constraint is that it is not permitted for both arguments to be consuming; for example, if the context node is a node in a streamed document, then the function call `f:min((price, discount))` would not be guaranteed streamable.

19.8.5.2 [Streamability Category: absorbing](#)

Informal description: Functions in this category typically read the subtrees rooted at the node or nodes supplied in the first argument. These subtrees must not overlap each other. The function must not return any streamed nodes.

Rules for the function signature: there are no constraints.

Rules for the function body: For the function to be [guaranteed-streamable](#), the [type-adjusted posture](#) of the function body with respect to the declared return type must be [grounded](#), and the [type-adjusted sweep](#) of the function body with respect to the declared return type must be [motionless](#) or [consuming](#).

Rules for references to the streaming parameter: If the declared type of the [streaming parameter](#) permits more than one node, then a variable reference referring to the streaming parameter is [striding](#) and [consuming](#). Otherwise such a variable reference is [striding](#) and [motionless](#).

Rules for function calls: If the first argument is [crawling](#) then the function call is [roaming](#) and [free-ranging](#); otherwise the [general streamability rules](#) apply. The operands are the expressions appearing in the argument list of the function call. The [operand usage](#) of the first argument is [absorption](#); the operand usage of other arguments is the [type-determined usage](#) based on the declared type of the corresponding [parameter](#) in the function signature.

Note:

Absorbing functions perform an operation analogous to atomization on their supplied arguments, in that they typically use information from the subtree rooted at a node to compute atomic values. Atomization can be seen as a special case of absorption. Calls on absorbing functions are therefore, from a streamability point of view, equivalent to calls on functions that implicitly atomize the supplied nodes.

An important difference, however, is that whereas atomization can be applied to any argument of a function call, absorption applies only to the first argument.

Another difference is that atomization is allowed on a sequence of nodes in [crawling](#) posture, whereas generalized absorption is not. Within a sequence, there may be nodes whose subtrees overlap, and the code for atomization is expected to handle this, but more general absorption operations are not. To write a function that accepts streamed nodes and atomizes them, it is better to use the streamability category [unclassified](#), and to declare the first argument with an atomic type, rather than using the category [absorbing](#) which allows more general processing, but restricts what can be supplied in the argument to the function call.

Example: An absorbing stylesheet function

The following function is declared as absorbing, and the function body meets the rules for this category because it makes downward selections only, and returns an atomic value.

```
<xsl:function name="f:count-descendants" as="xs:integer"
streamability="absorbing">
  <xsl:param name="input" as="node()*"/>
  <xsl:sequence select="count($input//*)"/>
</xsl:function>
```

The effect of the rules is that a call to this function is [guaranteed-streamable](#) provided that the sequence supplied as the value of the \$nodes argument is [motionless](#) or [consuming](#), and is either [grounded](#) or [striding](#).

Example: An absorbing stylesheet function with two arguments

The following function is declared as absorbing, and the function body meets the rules for this category because it makes downward selections only from the node supplied as the first argument, and returns an atomic value.

```
<xsl:function name="f:compare-size" as="xs:integer" streamability="absorbing">
  <xsl:param name="input0" as="node()"/>
  <xsl:param name="input1" as="node()"/>
  <xsl:sequence select="count($input0//*) - count($input1//*)"/>
</xsl:function>
```

This function takes two nodes as its arguments. Some examples of function calls include:

- Streamable: `f:compare-size(a, b)` where `a` is an element in a streamed document and `b` is an element in an unstreamed document
- Streamable: `f:compare-size(a, b)` where `a` and `b` are both elements in unstreamed documents
- Not streamable: `f:compare-size(a, b)` where `a` is an element in an unstreamed document and `b` is an element in a streamed document

The reason for the asymmetry is that for the first argument the operand usage is absorption, while for the second argument it is navigation. It is a consequence of the general streamability rules that when streamed nodes are supplied to an operand with usage navigation, the resulting expression is roaming and free-ranging.

Example: A recursive absorbing stylesheet function

The following function is declared as absorbing, and the function body meets the rules for this category. Analysis of the function body reveals that it is grounded and consuming; to establish this, it is necessary to analyze the recursive call `f:outline(*)`, and this is possible because it is known to be a call on an absorbing stylesheet function.

```
<xsl:function name="f:outline" as="xs:string" streamability="absorbing">
  <xsl:param name="input" as="element()*"/>
  <xsl:value-of select="$input ! (name() || '(' || f:outline(*) || ')')"
    separator=" , "/>
</xsl:function>
```

The effect of the rules is that a call to this function is guaranteed streamable in the typical case where the sequence supplied as the value of the `$input` argument is striding and consuming.

19.8.5.3 Streamability Category: inspection

Informal description: Functions in this category typically return properties of the node supplied in the first argument, where these properties can be determined without advancing the input stream. This allows access to properties such as the name and type of each node, and also to its ancestors, attributes, and namespaces.

Rules for the function signature: If the declared type of the streaming parameter permits more than one node, the function is not [guaranteed-streamable](#).

Rules for the function body: For the function to be [guaranteed-streamable](#), the [type-adjusted posture](#) of the function body with respect to the declared return type must be [grounded](#), and the [type-adjusted sweep](#) of the function body with respect to the declared return type must be [motionless](#).

Rules for references to the streaming parameter: Such a variable reference is [striding](#) and [motionless](#).

Rules for function calls: the [general streamability rules](#) apply. The operands are the expressions appearing in the argument list of the function call. The [operand usage](#) of the first argument is [inspection](#); the operand usage of other arguments is the [type-determined usage](#) based on the declared type of the corresponding argument in the function signature.

Note:

The [streaming parameter](#) is restricted to be a single node because if `$input` were a sequence of nodes, then an expression such as `($input/name(), $input/@id)` would not be streamable.

Example: Example of an inspection stylesheet function

The following function is declared with category [inspection](#), and the function body meets the rules for this category because all references to the supplied node are motionless.

```
<xsl:function name="f:depth" as="xs:integer" streamability="inspection">
  <xsl:param name="input" as="node()"/>
  <xsl:sequence select="count($input/ancestor-or-self::* )"/>
</xsl:function>
```

The effect of the rules is that a call to this function is guaranteed streamable provided that the expression supplied as the value of the `$nodes` argument is [motionless](#) or [consuming](#).

Example: Example of an inspection stylesheet function with two arguments

The following function is declared with category `inspection`, and the function body meets the rules for this category because the function signature ensures that the second argument cannot be a node.

```
<xsl:function name="f:get-attribute-value" as="xs:string">
  <xsl:param name="element" as="node()"/>
  <xsl:param name="attribute-name" as="xs:string"/>
  <xsl:sequence select="string($element/@*[local-name() = $attribute-name])"/>
</xsl:function>
```

Although the normal usage of this function might be to supply an element from a streamed document as the first argument, and a literal string as the second, it is also permissible (and guaranteed streamable) to supply an unstreamed element as the first argument, and an element node from a streamed document as the second. When applying the general streamability rules in this case, the first operand is grounded and motionless, while the second is grounded and consuming (by virtue of the rules for type-determined usage), and this makes the function call grounded and consuming.

19.8.5.4 Streamability Category: filter

Informal description: Functions in this category typically return either the node supplied in the first argument or nothing, depending on the values of properties that can be determined without advancing the input stream. This allows access to properties such as the name and type of each node, and also to its ancestors, attributes, and namespaces.

Rules for the function signature: If the declared type of the streaming parameter permits more than one node, the function is not guaranteed-streamable.

Rules for the function body: For the function to be guaranteed-streamable, the type-adjusted posture of the function body with respect to the declared return type must be striding, and the type-adjusted sweep of the function body with respect to the declared return type must be motionless.

Rules for references to the streaming parameter: Such a variable reference is striding and motionless.

Rules for function calls: The posture and sweep of a call to a function in this category are determined by applying the general streamability rules. The operands are the expressions supplied as arguments to the function call. The first argument has operand usage transmission; any further arguments have type-determined usage based on the declared type of the corresponding parameter in the function signature.

Example: Example of a filtering stylesheet function

The following function is declared as filtering, and the function body meets the rules for this category because it selects nodes from the input based on motionless properties (namely, the existence of attributes).

```
<xsl:function name="f:large-regions" as="element(region)"
streamability="filter">
  <xsl:param name="input" as="element(region)" />
  <xsl:sequence select="$input[@size gt 1000]" />
</xsl:function>
```

The effect of the rules is that the posture and sweep of a function call `f:large-regions(EXPR)` are the same as the posture and sweep of `EXPR`.

Although the name `filter` suggests that the result must always be a subset of the input, this is not strictly required by the rules. The function can also return atomic values, as well as attribute and namespace nodes.

19.8.5.5 Streamability Category: shallow-descent

Informal description: Functions in this category typically return children of the nodes supplied in the first argument. They may also select deeper in the subtrees of these nodes, provided that no node in the result can possibly be an ancestor of any other node in the result.

Rules for the function signature: If the declared type of the streaming parameter permits more than one node, the function is not [guaranteed-streamable](#).

Rules for the function body: For the function to be [guaranteed-streamable](#), the [type-adjusted posture](#) of the function body with respect to the declared return type must be [striding](#), and the [type-adjusted sweep](#) of the function body with respect to the declared return type must be [motionless](#) or [consuming](#).

Rules for references to the streaming parameter: Such a variable reference is [striding](#) and [motionless](#).

Rules for function calls: The rules are as follows, in order:

1. Let T_0 be the [U-type](#) corresponding to the declared type of the [streaming parameter](#) in the function signature (defaulting to $U\{*\}$).
2. Let P_0 and S_0 be the [type-adjusted posture and sweep](#) of the first argument expression, based on type T_0 .
3. If P_0 is not [striding](#) or [grounded](#), the function call is [roaming](#) and [free-ranging](#).
4. Consider a construct C whose operands are the argument expressions other than the first argument, with [type-determined operand usage](#) based on the declared type of the corresponding parameter in the function signature. Let P_1 and S_1 be the [posture](#) and [sweep](#) of C , assessed using the [general streamability rules](#).

Note:

If there is only one argument, then P_1 is [grounded](#) and S_1 is [motionless](#).

5. If P_1 is not [grounded](#), the function call is [roaming](#) and [free-ranging](#).

6. If S_0 and S_1 are both consuming, or if either is free-ranging, then the function call is roaming and free-ranging.
7. If P_0 is grounded, then the posture of the function call is grounded, and the sweep of the function call is the wider of S_0 and S_1 .
8. Otherwise, the posture of the function call is P_0 , and the sweep of the function call is as follows:
 - a. If the intersection of T_0 with $U\{document-node(), element()\}$ is empty (that is, the declared type of the first argument does not permit document or element nodes) then S_0 .
 - b. Let A be the static type of the expression supplied as the first argument. If the intersection of A with $U\{document-node(), element()\}$ is empty (that is, the inferred type of the expression supplied as the first argument does not permit document or element nodes) then S_0 .
 - c. Otherwise, consuming.

Example: A shallow-descent stylesheet function

The following function is declared as shallow-descent, and the function body meets the rules for this category because it selects children of the supplied input node.

```
<xsl:function name="f:alternate-children" as="node()*"
               streamability="shallow-descent">
  <xsl:param name="input" as="element()"/>
  <xsl:sequence select="$input/node()[position() mod 2 = 1]"/>
</xsl:function>
```

The effect of the rules is that a call to this function is guaranteed streamable in the typical case where the node supplied as the value of the \$input argument is striding and consuming.

19.8.5.6 Streamability Category: deep-descent

Informal description: Functions in this category typically return descendants of the nodes supplied in the first argument.

Rules for the function signature: If the declared type of the streaming parameter permits more than one node, the function is not guaranteed-streamable.

Rules for the function body: For the function to be guaranteed-streamable, the type-adjusted posture of the function body with respect to the declared return type must be crawling, and the type-adjusted sweep of the function body with respect to the declared return type must be motionless or consuming.

Rules for references to the streaming parameter: Such a variable reference is striding and motionless.

Rules for function calls: The rules are as follows, in order:

1. Let T_0 be the U-type corresponding to the declared type of the streaming parameter in the function signature (defaulting to $U\{*\}$).
2. Let P_0 and S_0 be the type-adjusted posture and sweep of the first argument expression, based on type T_0 .
3. If P_0 is not striding or grounded, the function call is roaming and free-ranging.

4. Consider a construct C whose operands are the argument expressions other than the first argument, with type-determined operand usage based on the declared type of the corresponding parameter in the function signature. Let P_1 and S_1 be the posture and sweep of C , assessed using the general streamability rules

Note:

If there is only one argument, then P_1 is grounded and S_1 is motionless.

5. If P_1 is not grounded, the function call is roaming and free-ranging.
6. If S_0 and S_1 are both consuming, or if either is free-ranging, the function call is roaming and free-ranging.
7. If P_0 is grounded, then the posture of the function call is grounded, and the sweep of the function call is the wider of S_0 and S_1 .
8. Otherwise, the posture of the function call is crawling, and the sweep of the function call is as follows:
 - a. If the intersection of T_0 with $U\{document-node(), element()\}$ is empty (that is, the declared type of the first argument does not permit document or element nodes) then S_0 .
 - b. Let A be the static type of the expression supplied as the first argument. If the intersection of A with $U\{document-node(), element()\}$ is empty (that is, the inferred type of the expression supplied as the first argument does not permit document or element nodes) then S_0 .
 - c. Otherwise, consuming.

Example: A deep-descent stylesheet function

The following function is declared as deep-descent, and the function body meets the rules for this category because it selects descendants of the supplied input node.

```
<xsl:function name="f:all-comments" as="comment()*"
               streamability="deep-descent">
  <xsl:param name="input" as="element()"/>
  <xsl:sequence select="$input//comment()" />
</xsl:function>
```

The effect of the rules is that a call to this function is guaranteed streamable in the typical case where the node supplied as the value of the `$input` argument is striding and consuming.

19.8.5.7 Streamability Category: ascent

Informal description: Functions in this category typically return ancestors of the nodes supplied in the first argument.

Rules for the function signature: If the declared type of the streaming parameter permits more than one node, the function is not guaranteed-streamable.

Rules for the function body: For the function to be guaranteed-streamable, the type-adjusted posture of the function body with respect to the declared return type must be either climbing or grounded, and the type-adjusted sweep of the function body with respect to the declared return type must be motionless.

Rules for references to the streaming parameter: Such a variable reference is climbing and motionless.

Rules for function calls: The posture and sweep of a call to a function in this category are determined as follows:

1. Let P_0 and S_0 be the posture and sweep obtained by assessing the function call using the general streamability rules, where the operands are the arguments to the function call, with an operand usage for the first argument of inspection, and an operand usage for arguments after the first being the type-determined usage based on the declared type of the corresponding function parameter.
2. If P_0 is roaming or S_0 is free-ranging, then the function call is roaming and free-ranging.
3. If S_0 is not motionless, then the function call is roaming and free-ranging.
4. If P_0 is roaming, then the function call is roaming and free-ranging.
5. If P_0 is grounded, then the function call is grounded and motionless.
6. Otherwise, the function call is climbing and motionless.

Example: An ascending stylesheet function

The following function is declared with category ascent, and the function body meets the rules for this category because it selects ancestors of the supplied node.

```
<xsl:function name="f:containing-section" as="element(section)"
               streamability="ascent">
  <xsl:param name="input" as="element(para)*"/>
  <xsl:sequence select="$input/ancestor::section[last()]"/>
</xsl:function>
```

The effect of the rules is that a call to this function is guaranteed streamable provided that the node supplied as the value of the input argument is not roaming or free-ranging. There are no other constraints on the node supplied in the input sequence.

19.8.6 Classifying Attribute Sets

The posture of an attribute set is always grounded (its result can never return streamed nodes).

The sweep of an attribute set is motionless if all the following conditions hold:

1. Every xsl:attribute instruction within the declarations comprising the attribute set is motionless when assessed as described in [10.2.3 Streamability of Attribute Sets](#), using a context posture of striding.
2. Every attribute set referenced in the use-attribute-sets attribute of an xsl:attribute-set declaration of the attribute set has the attribute streamable="yes".
- 3.

If the sweep of an attribute set is not motionless then it is free-ranging.

Note:

Attribute sets will always be [grounded](#), because they return newly constructed attribute nodes.

Attribute sets will very often be [motionless](#), but if they access the context item, they may be [free-ranging](#). Although some attribute sets could theoretically be classified as [consuming](#), this option has been excluded because it is unlikely to be useful; given the requirement to create attributes whose values are obtained by reading a streamed input document, use of a streamable [template rule](#) is a more versatile approach.

Because attribute sets can be overridden in another [package](#), the streamability of a construct such as an [xsl:element](#) instruction containing a [use-attribute-sets](#) attribute is based on the declared streamability of the named attribute sets, as defined by the [streamable](#) attribute of the [xsl:attribute-set](#) element. If [streamable="yes"](#) is specified, then there is a requirement that any overriding attribute set should also specify [streamable="yes"](#), and a streaming processor is required to check that an attribute set containing such a declaration does in fact satisfy the streamability rules.

19.8.7 [Classifying Value Templates](#)

A [value template](#) (that is, an [attribute value template](#) or [text value template](#)) is a [construct](#) whose operands are the expressions contained within curly brackets. The required type for this operand role is [xs:string](#) and the [usage](#) is [absorption](#).

The [sweep](#) and [posture](#) of a value template are determined using the general rules in [19.8.1 General Rules for Streamability](#).

If there are no expressions contained within curly brackets, the value template is [motionless](#).

19.8.8 [Classifying Expressions](#)

XPath expressions are classified using the rules in this section.

In the analysis that follows, [expressions](#) are classified according to the most specific production rule that they match for which there is an entry in this section. A production P is considered more specific than a production Q ($Q \neq P$) if every expression that matches P also matches Q . For example:

- The expression `3` satisfies the productions `NumericLiteral`, `Literal`, and `ArithmeticExpression`; the most specific of these for which there is an entry in this section is `Literal`.
- The expression `text()` (appearing as an expression) is a `TextTest`, and therefore a `KindTest`, which is itself a `NodeTest`, and therefore an `AxisStep` with a defaulted `ForwardAxis`. The most specific of these for which there is an entry in this section is `AxisStep`. Although the expression is also a `RelativePathExpr`, that production is less specific than `AxisStep` so its rules do not apply.
- The expression `section/title` is a `RelativePathExpr`, for which there is an entry in this section. Although the expression is also a `PathExpr`, that production is less specific than `RelativePathExpr` so its rules do not apply.

The production rules for different kinds of expression are listed (with their names and numbers) in the order in which they appear in Appendix A.1 of the XPath 3.0 specification; rules are also given for new constructs

introduced by XPath 3.1. Where two numbers are given, they are the production rule numbers in XPath 3.0 and XPath 3.1 respectively; where there is a single number, it is the production rule number in XPath 3.1.

Many expressions can be analyzed using the [general streamability rules](#). These are indicated in the table below by means of a simple proforma in which the [operand roles](#) are represented by a short code (A = [absorption](#), I = [inspection](#), T = [transmission](#), N = [navigation](#)). For example the proforma A + A indicates that for an arithmetic expression, both operands have [operand usage absorption](#), while I or I indicates that for an or expression, both operands have [operand usage inspection](#). For expressions where further explanation is needed, the table contains a link to the relevant section.

Operand Roles for XPath Expressions

Construct	Proforma or Reference to Detailed Rules	Further Information
Expr [6,6]	T, T	
ForExpr [8,8]	See 19.8.8.1 Streamability of for Expressions	
LetExpr [11,11]	let \$var := N return T	Binding of variables to streamed nodes is not allowed.
QuantifiedExpr [14,14]	See 19.8.8.2 Streamability of Quantified Expressions	
IfExpr [15,15]	if (I) then T else T	The then-clause and else-clause form a choice operand group with usage transmission
OrExpr [16,16]	I or I	
AndExpr [17,17]	I and I	
StringConcatExpr [19,19]	A A	
RangeExpr [20,20]	A to A	
AdditiveExpr [21,21]	A + A, A - A	
MultiplicativeExpr [22,22]	A * A, A div A, etc.	
UnionExpr [23,23]	See 19.8.8.4 Streamability of union, intersect, and except Expressions	
IntersectExceptExpr [24,24]	See 19.8.8.4 Streamability of union, intersect, and except Expressions	

Construct	Proforma or Reference to Detailed Rules	Further Information
InstanceOfExpr [25,25]	See 19.8.8.5 Streamability of instance of Expressions	
TreatExpr [26,26]	See 19.8.8.6 Streamability of treat as Expressions	
CastableExpr [27,27]	A <code>castable</code> as TYPE	
CastExpr [28,28]	A <code>cast</code> as TYPE	
UnaryExpr [29,30]	+A, -A	
GeneralComp [31,32]	A = A, A < A, A != A, etc.	
ValueComp [32,33]	A eq A, A lt A, A ne A, etc.	
NodeComp [33,34]	I is I, I << I, I >> I	See Note 1 below
SimpleMapExpr [34,35]	See 19.8.8.7 Streamability of Simple Mapping Expressions	
PathExpr [35,36]	See 19.8.8.8 Streamability of Path Expressions	
RelativePathExpr [36,37]	See 19.8.8.8 Streamability of Path Expressions	
AxisStep [38,39]	See 19.8.8.9 Streamability of Axis Steps	
ForwardStep [39,40], ReverseStep [42,43]	See 19.8.8.9 Streamability of Axis Steps	
PostfixExpr [48,49]: Filter Expression	See 19.8.8.10 Streamability of Filter Expressions	
PostfixExpr [48,49]: Dynamic Function Call	See 19.8.8.11 Streamability of Dynamic Function Calls	
Literal [53,57]		There are no operands, so the construct is <u>grounded</u> and <u>motionless</u>
VarRef [55,59]	See 19.8.8.12 Streamability of Variable References	
ParenthesizedExpr [57,61]	(T)	
	()	There are no operands, so the construct is <u>grounded</u> and <u>motionless</u>

Construct	Proforma or Reference to Detailed Rules	Further Information
ContextItemExpr [58,62]	See 19.8.8.13 Streamability of the Context Item Expression	
FunctionCall [59,63]	See 19.8.8.14 Streamability of Static Function Calls	
NamedFunctionRef [63,67]	See 19.8.8.15 Streamability of Named Function References	
InlineFunctionExpr [64,68]	See 19.8.8.16 Streamability of Inline Function Declarations	
MapConstructor [-,69]	See 19.8.8.17 Streamability of Map Constructors	
Lookup (Postfix [-,49] and Unary [-,53])	See 19.8.8.18 Streamability of Lookup Expressions	
ArrowExpr [-,29]	See 19.8.8.14 Streamability of Static Function Calls and 19.8.8.11 Streamability of Dynamic Function Calls : the rules for $X \Rightarrow F(Y, Z)$ are the same as the rules for $F(X, Y, Z)$	
SquareArrayConstructor [-,74]	[N, N, ...]	
CurlyArrayConstructor [-,75]	array{N, N, ...}	

Note:

1. The operators `is`, `<<`, and `>>` apply to streamed nodes just as to any other nodes, though there are few practical situations where they will be useful. A streamed document conforms to the rules of the XDM data model, and its nodes are therefore distinct and ordered. They follow the usual rules, for example that a parent node precedes its children in document order. Expressions such as `.. is parent::X` or `ancestor::x[1] << ancestor::y[1]` are therefore perfectly meaningful. The usefulness of the operators is limited by the fact that variables cannot be bound to nodes in a streamed document. It is permitted, though perhaps not useful, for one of the operands to be `consuming`: one can write `. << child::x`, and the resulting expression is (by applying the general rules) `consuming` and grounded.

The restriction that variables cannot be bound to streamed nodes prevents writing of expressions such as `let $x := . return descendant::x[ancestor::y[1] is $x]`. As a workaround, the intended effect can be achieved by comparing node identity using the `generate-id`^{FO30} function: `let $x := generate-id(.) return descendant::x[generate-id(ancestor::y[1]) = $x]`

19.8.8.1 Streamability of for Expressions

Writing the expression as `for $v in S return R`, the two operand roles are *S* and *R*.

The posture and sweep are determined by the first of the following that applies:

1. If *S* is not grounded, then roaming and free-ranging.
2. Otherwise, the general streamability rules apply. The operand roles are:
 - a. The `in` expression (*S*). This has usage navigation.
 - b. The `return` expression (*R*). This is a higher-order operand with usage transmission.

Note:

Expressions of the form `for $i in 1 to 3 return $i*2`, where there is no reference to a streamed node, are clearly streamable.

The `in` expression can also be consuming, for example `for $e in copy-of(emp) return $e/salary`.

The rule that *S* must be grounded prevents the variable being bound to a node in a streamed document. This disallows expressions of the form `for $x in child::section return $x/para`, because this requires data flow analysis (tracing from the binding of a variable to its usages), rather than purely syntactic analysis. Some implementations may be able to stream such constructs.

The fact that the `return` clause is a higher-order operand prevents it from being a consuming expression, for example `for $i in 1 to 3 return salary`. Use of a motionless expression that accesses streamed nodes is however allowed, for example `for $i in 1 to 3 return name(ancestor::x[$i])`.

19.8.8.2 Streamability of Quantified Expressions

An expression with multiple `in`-clauses is first rewritten using nested quantified expressions: for example `some $i in X, $j in Y satisfies $i eq $j` can be rewritten as `some $i in X satisfies (some $j in Y satisfies $i eq $j)`. The analysis therefore only needs to consider expressions with a single `in`-clause.

Writing such an expression as `some|every $v in S satisfies C`, the two operand roles are *S* and *C*.

The general streamability rules apply. The operand roles are:

1. The `in` expression (*S*). This has usage navigation.
2. The `satisfies` expression (*C*). This is a higher-order operand with usage inspection.

Note:

Expressions of the form `some $i in 1 to 3 satisfies $i lt 2`, where there is no reference to a streamed node, are clearly streamable.

The expression `S` can be [consuming](#), so long as it is grounded: for example `some $e in emp/salary/number(.) satisfies $e gt 10000`.

The rule that `S` has usage [navigation](#) prevents the variable being bound to a node in a streamed document. This disallows expressions of the form `some $x in child::section satisfies has-children($x)`, because this requires data flow analysis (tracing from the binding of a variable to its usages), rather than purely syntactic analysis. Some implementations may be able to stream such constructs.

The fact that `C` is a higher-order operand prevents it from being a [consuming](#) expression: for example `some $i in 1 to 3 satisfies author[$i] eq "Kay"` is not streamable. Use of a motionless expression that accesses streamed nodes is however allowed, for example `some $i in 1 to 3 satisfies @grade = $i`.

Quantified expressions that fail the streamability rules can often be rewritten as filter expressions. For example, the expression `some $x in child::section satisfies has-children($x)` can be rewritten as `exists(child::section[has-children(.)])`, which is grounded and [consuming](#).

[19.8.8.3 Streamability of if expressions](#)

Writing the expression as `if (C) then T else E`, there are three operand roles: `C`, `T`, and `E`. The [usage](#) of `C` is [inspection](#), while the [usage](#) of `T` and `E` is [transmission](#). Operands `T` and `E` form a [choice operand group](#), meaning that they can both consume the input stream, provided they have consistent [posture](#). The [general streamability rules](#) apply.

[19.8.8.4 Streamability of union, intersect, and except Expressions](#)

The [posture](#) and [sweep](#) are the first of the following that applies:

1. If either of the two operands is [free-ranging](#), then [roaming](#) and [free-ranging](#) (Example: `. | following-sibling:::*`).
2. If either of the two operands is [grounded](#) and [motionless](#), then the [posture](#) and [sweep](#) of the other operand (Example: `. | doc('abc.com')//x`)
3. If both operands are [climbing](#), then [climbing](#) and the wider of the sweeps of the two operands (Example: `parent::A | */ancestor::B`).
4. If the left-hand operand is [striding](#) or [crawling](#) and the right-hand operand is also [striding](#) or [crawling](#), then [crawling](#) and the wider of the sweeps of the two operands (Example: `* | */*`).
5. Otherwise, [roaming](#) and [free-ranging](#) (Example: `child::div | parent::div`).

Note:

Essentially the principle is that if both operands are streamable, then the result is streamable (this assumes an evaluation strategy where both operands are evaluated during the same pass of the streamed input document, and the results merged). But there are caveats because of the need for static streamability analysis of the result. This prevents constructs such as `.. | *` that have heterogeneous [posture](#).

Where the two operands are both [striding](#), there are cases where an implementation could determine that the result is also [striding](#): for example `(author | editor)`. In general, however, the combination of two striding operands may produce a sequence of nodes that have nested subtrees (consider `author | author/name`), so the result is classified as [crawling](#).

The expression `(author | editor)`, although it is not [striding](#), can be rewritten in the form `* [self::author or self::editor]`, which is [striding](#).

19.8.8.5 [Streamability of instance of Expressions](#)

For an expression of the form X instance of ST (where X is an expression and ST is a [SequenceType](#)), the [posture](#) and [sweep](#) are determined by the [general streamability rules](#). There is a single operand X , whose [operand usage](#) is as follows:

1. If the [ItemType](#) of ST is a [DocumentTest](#), optionally parenthesized, that contains an [ElementTest](#) or [SchemaElementTest](#) then absorption
2. Otherwise, inspection.

Note:

In general, it is possible to determine whether a node matches an [ItemType](#) without consuming the node. For example it can be established whether an element matches the test `element(para)` when positioned at the start tag.

An [ItemType](#) of the form `document-node(element(X))` is a exception to this rule because it matches a document node only if it has exactly one element node child, and this cannot be determined without consuming the document.

A processor may have knowledge that the document node cannot contain multiple element nodes, for example because it knows that the source of the streamed document is an XML parser that is not capable of generating such a stream. In such cases the processor may make a different assessment of the streamability of this construct. This comes under the general provision that a processor is always at liberty to use streaming even when the stylesheet is not guaranteed streamable.

Note:

As with other constructs that are evaluated with inspection usage, for example the `nameFO30` function or access to an attribute node, evaluation of a construct such as `$X instance of schema-element(E)` as true or false may be invalidated if reading of the input stream subsequently fails. Dynamic errors during streamed processing of an input document invalidate all output generated prior to the failure, and this case is no different.

Note:

Given an expression such as `child::* instance of element(E)*`, the expression as a whole is consuming and grounded. By contrast, the expression `. instance of element(E)*` is motionless and grounded. This can be verified by applying the general streamability rules to these cases.

19.8.8.6 Streamability of treat as Expressions

For an expression of the form `X treat as ST` (where `X` is an expression and `ST` is a SequenceType), the posture and sweep are determined as follows:

1. If the ItemType of `ST` is a DocumentTest, optionally parenthesized, that contains an ElementTest or SchemaElementTest then roaming and free-ranging.
2. Otherwise, the general streamability rules apply. There is a single operand `X`, whose operand usage is transmission.

Note:

See the notes in [19.8.8.5 Streamability of instance of Expressions](#) for a discussion of the streamability difficulties associated with `document-node()` tests.

19.8.8.7 Streamability of Simple Mapping Expressions

The mapping operator `!` is treated as a left-associative binary operator, so the expression `a!b!c` is processed as `(a!b)!c`.

The posture of the expression is the posture of the right-hand operand, assessed with a context posture and type set to the posture and type of the left-hand operand.

The sweep of the expression is the wider of the sweeps of the two operands.

19.8.8.8 Streamability of Path Expressions

The streamability analysis applies after the expansion of the `//` pseudo-operator to `/descendant-or-self::node()`, and after expanding `..` to `parent::node()`, `@X` to `attribute::X`, and an omitted axis to the

default axis for the node kind.

Following the rules in XPath, a leading "/" is converted to `(root(self::node()) treat as document-node())/` (with the final "/" omitted for the expression "/" on its own). This is followed by a rewrite of the call on `root`^{FO30}, as described in [19.8.9.18 Streamability of the root Function](#).

Note:

Taken together, these rewrites have the effect that a path expression such as `//a` is streamable only if the statically-determined context item type is `document-node()`, which will be the case for example immediately within `xsl:source-document`, or in a template rule with `match="/"`.

A `RelativePathExpr` with more than two operands (such as `a/b/c`) is taken as a tree of binary expressions (that is, `(a/b)/c`).

The `sweep` of a relative path expression is the wider `sweep` of the two operands, where the ordering of increasing width is `motionless, consuming, free-ranging`.

Note:

Examples:

- The `sweep` of `a/@code` is `consuming` (the wider of `consuming` and `motionless`).
- The `sweep` of `a/descendant::b` is `consuming` (the wider of `consuming` and `consuming`).
- The `sweep` of `./@code` is `motionless` (the wider of `motionless` and `motionless`).
- The `sweep` of `./a` is `consuming` (the wider of `motionless` and `consuming`).
- The `sweep` of `a/following::b` is `free-ranging` (the wider of `consuming` and `free-ranging`).
- The `sweep` of `./.` is `motionless` (the wider of `motionless` and `motionless`).

The `posture` of a relative path expression is assessed in two phases, as follows:

1. First, the provisional `posture` is determined as follows: The provisional `posture` of the expression is the `posture` of the right-hand operand, assessed with a `context posture` and type set to the `posture` and type of the left-hand operand; and the provisional sweep is the wider of the sweeps of the two operands.
2. If the provisional `posture` is `roaming`, then it is reassessed as follows:
 - a. [DEFINITION: A `RelativePathExpr` is a **scanning expression** if and only if it is syntactically equivalent to some `motionless pattern`.]

Note:

This means that a `RelativePathExpr` is a **scanning expression** if it conforms to the grammar for a `RelativePathExprP` in the grammar for patterns (see [5.5.2 Syntax of Patterns](#)), and if, when considered as a pattern, the pattern is motionless according to the rules in [19.8.10 Classifying Patterns](#).

In practice, the test as to whether the construct is equivalent to a pattern is likely to be made by examining the structure of the expression tree, rather than by re-parsing the lexical form of the expression against the grammar for patterns; but the outcome is the same.

b. If the expression is a **scanning expression** then:

- i. If the static type of the expression contains `U{element}` then its [posture](#) is [crawling](#).
- ii. Otherwise, its [posture](#) is [striding](#)

3. Otherwise (if the provisional [posture](#) is not [roaming](#), or the expression is not a **scanning expression**), the [posture](#) of the expression is the provisional [posture](#).

Note:

The special rules for scanning expressions are designed to ensure that expressions such as `//section/head` are streamable. The problem with such an expression is that it is possible to have two nested sections *A* and *B*, where *A* is the parent of *B* and thus precedes *B* in document order, but where there are children of *A* that come *after* children of *B* in document order. This means that a nested-loop strategy for the evaluation of `/descendant::section/child::head` is not guaranteed to deliver nodes in document order without a sort, and is therefore not a viable strategy for streaming.

However, there is a different strategy for evaluating such an expression, which is in effect to rewrite the expression as `/descendant::head[parent::section]`; specifically, it is possible to scan all descendants in document order, looking for a `head` element that has a `section` parent. Hence the term **scanning expressions**.

The expressions that qualify as scanning expressions are paths that can be evaluated by scanning all descendants and testing each one (independently) to see whether the elements on its ancestor axis match the specified path. The subset of expressions that qualify as scanning expressions is therefore the same as the subset that qualify as motionless patterns.

Scanning expressions cannot use positional predicates: for example `//section/head[1]` is not recognized as a scanning expression because this would require information about a streamed node (specifically, about its preceding siblings) that is not retained during streaming.

Note:

Perhaps surprisingly, the expression `./section/head` is not a scanning expression and is therefore not guaranteed streamable. This is because it does not take the syntactic form of a [pattern](#). To make it streamable, it can be rewritten as `descendant::section/head` or as `self::node()//section/head`.

Similarly, within a streamable stylesheet function whose [streaming parameter](#) is `$node`, the expression `$node//section/head` is not a scanning expression. In this case the expression does have the syntactic form of a pattern, but the pattern is not classified as motionless. (See [19.8.10 Classifying Patterns](#) — a motionless pattern cannot contain a `RootedPath`.) A workaround in this case is to rewrite the expression as `$node/(descendant::section/head)`. Assuming that the function in question declares `streamability="absorbing"`, the analysis here is that the left-hand operand (`$node`) is striding and consuming, while the right hand operand (`descendant::section/head`) is crawling and consuming (because it is a scanning expression). The expression as a whole is therefore crawling and consuming.

These are cases where an implementation might reasonably choose to relax the rules, insofar as this is permitted by [19.10 Streamability Guarantees](#).

Note:

Examples:

In each of the following cases, assume that the [context posture](#) is striding.

- The [posture](#) of the expression `a/b/c` is striding, because (under the rules for AxisStep [38]) a child axis step evaluated with striding context [posture](#) creates a new striding posture.
- The posture of the expression `a/descendant::c` is crawling, because a descendant axis step evaluated with striding context posture creates a new crawling posture.
- The posture of the expression `../@status` is striding, because a parent axis step evaluated with striding context posture creates a new climbing posture, and an attribute axis step evaluated with climbing context posture creates a new striding posture.
- The posture of the expression `copy-of(.)//a/following-sibling::*[1]` is grounded, because the [copy-of](#) evaluated with striding posture creates a grounded posture, and all subsequent axis steps leave this posture unchanged.
- The expression `section//head` expands to `(section/descendant-or-self::node())/child::head`. The posture of the left-hand operand `section/descendant-or-self::node()` is crawling, because a descendant axis step evaluated with striding context posture creates a new crawling posture. The provisional posture of the expression as a whole is therefore [roaming](#), because a child axis step evaluated with crawling context posture gives a resulting roaming posture. However, the expression is a scanning expression (both `section//head` and its expansion are motionless patterns), so the expression as a whole has crawling posture.
- The expression `section//head[1]` is free-ranging: unlike the previous example, it contains a positional predicate, which means that the operands do not satisfy the rules for scanning expressions.

19.8.8.9 *Streamability of Axis Steps*

The sweep and posture of an AxisStep S are determined by the first of the following rules that applies:

1. If the context posture is grounded, then the sweep is motionless and the posture is grounded;
 2. If the context posture is roaming, then the sweep is free-ranging and the posture is roaming;
 3. If the statically-inferred context item type is such that the axis will always be empty (for example, applying the child axis to a text node or the parent axis to a document node), or if the NodeTest is one that can never select nodes on the chosen axis (for example, selecting attribute nodes on the child axis), then the sweep is motionless and the posture is grounded (because the expression is statically known to return an empty sequence);
 4. If all the following conditions are satisfied:
 - a. The context posture is striding
 - b. The axis is descendant or descendant-or-self
 - c. There is a predicate P in the PredicateList that satisfies all the following conditions:
 - i. The static type of P is a subtype of $U\{xs:decimal, xs:double, xs:float\}$
 - ii. Neither P , nor any operand of P , at any depth provided it has the AxisStep S as its focus-setting container, is a context item expression, an axis expression, or a call on a focus-dependent function;
- then striding and consuming

Note:

Examples are `descendant::section[1]`, `descendant::section[$i+1]`, `descendant::section[count($x)]`. The significance of this rule is that it detects cases where the descendant axis selects a singleton, and where the posture of the result can therefore be striding rather than crawling.

5. If the PredicateList contains a Predicate that is not motionless, then the sweep is free-ranging and the posture is roaming;
6. Otherwise, the sweep and posture of the expression are as determined by the table below, based on the context posture, the choice of axis, and the node test. The condition “Selects elements?” is true if the U-type of S has a non-empty intersection with $U\{element()\}$.

Streamability of Axis Steps Based on Context Posture

Context posture	Axis	Selects elements?	Result posture	Sweep
Grounded	any		Grounded	Motionless
Climbing	self, parent, ancestor-or-self, ancestor		Climbing	Motionless
Climbing	attribute, namespace		Striding	Motionless
Striding	parent, ancestor-or-self, ancestor		Climbing	Motionless

Context posture	Axis	Selects elements?	Result posture	Sweep
Striding	self, attribute, namespace		Striding	Motionless
Striding	child		Striding	Consuming
Striding	descendant, descendant-or-self	Yes	Crawling	Consuming
Striding	descendant, descendant-or-self	No	Striding	Consuming
Crawling	parent, ancestor-or-self, ancestor		Climbing	Motionless
Crawling	attribute, namespace		Striding	Motionless
Crawling	self	Yes	Crawling	Motionless
Crawling	self	No	Striding	Motionless
Any other combination			Roaming	Free-ranging

Note:

This analysis does not attempt to classify `para[title]` as a [consuming](#) expression; an implementation might choose to do so.

19.8.8.10 [Streamability of Filter Expressions](#)

For a filter expression F of the form $B[P]$ (where B might itself be a filter expression), the [posture](#) and [sweep](#) are the first of the following that applies:

1. If all the following conditions are satisfied:
 - a. B is crawling;
 - b. The static type of P is a subtype of $\{xs:decimal, xs:double, xs:float\}$, and
 - c. Neither P , nor any operand of P , at any depth provided it has F as its focus-setting container, is a context item expression, an axis expression, or a call on a focus-dependent function

then the [posture](#) is [striding](#) and the [sweep](#) is the sweep of B .

Note:

This rule captures cases where it can be statically determined that the predicate is numeric and is independent of the focus. In such cases, the filter expression selects at most one node, and the posture can therefore be changed from crawling to striding (if there is only one node, there can be no overlapping trees). Examples of filter expressions that satisfy this test are `(//x)[3]`, `(//x)[\$i+1]`, `(//x)[index-of($a, $b)[last()]]`, and `(//x)[1 to 5]`. The last example will actually raise a type error because `1 to 5` has no effective boolean value; but if expressions are going to fail, it does not matter what their streamability properties are.

2. If P is [motionless](#), then the [posture](#) and [sweep](#) of B ;

Note:

This includes the case where B is grounded. The predicate P is assessed with the posture of B as its context posture, and if this is grounded, then P will almost invariably be motionless, making the filter expression as a whole grounded and motionless. For example if $\$s$ is grounded, then $\$s[child::*]$ is also grounded. A counter-example is the expression $\$s[$n = 2]$ where $\$n$ is a reference to the first argument of a stylesheet function that is [declared-streamable](#): here the predicate is not motionless, so the filter expression is roaming and free-ranging.

3. Otherwise, [roaming](#) and [free-ranging](#).

Note:

The first rule allows a construct such as `<xsl:apply-templates select="(//title)[1]" />`, where a [crawling](#) operand would not be guaranteed streamable.

Note:

This section is not applicable to predicates forming part of an axis step, such as `//title[1]`, as these are not technically filter expressions. See [19.8.8.9 Streamability of Axis Steps](#).

19.8.8.11 [Streamability of Dynamic Function Calls](#)

Note:

This section applies to dynamic function calls written using the traditional syntax `$F(X, Y, Z)` and equally to those using the new XPath 3.1 syntax `X => $F(Y, Z)`

The [posture](#) and [sweep](#) of a dynamic function call such as `$F(X, Y)` are determined by the [19.8.1 General Rules for Streamability](#). The operands and their usages are as follows:

1. The base expression that computes the function value itself (here `$F`). This has usage [inspection](#).
2. The argument expressions excluding any ? placeholders (here `X` and `Y`). These have [type-determined usage](#) dependent on ancillary information associated with the [static type](#) of the base expression, where available (see

[19.1 Determining the Static Type of a Construct](#)). If this information indicates that the base expression is a function with signature `function(A, B, ...) as R`, then the first argument X has [type-determined usage](#) based on the first argument type A, the second argument Y has [type-determined usage](#) based on the second argument type B, and so on. If no function signature is available, then the usage of each of the argument expressions is [navigation](#).

Note:

As explained in [10.3.6 Dynamic Access to Functions](#), use of a dynamic function call where the function value is bound to a focus-dependent function such as `name#0`, `lang#1`, or `last#0` is likely to lead to a dynamic error if the context item is a node in a streamed document, but this does not affect the static streamability analysis.

Note:

Maps and arrays are functions, and it is possible to look up a value in a map or array using a dynamic function call of the form `$map($key)` or `$array($index)`. If it is statically known that the function in question is a map or array, then it is also known that the argument type is `xs:anyAtomicType`, and that the operand usage is therefore [absorption](#). A call that passes a streamed node will therefore be [grounded](#) and [consuming](#). However, if it is not known statically that the function is a map or array, then the expression will generally be [roaming](#) and [free-ranging](#).

This means it is desirable to declare the type of any variable holding a map or array. If streamable nodes are used to lookup a value in a map or array, then it may be advisable to use the `map:get` or `array:get` functions explicitly; or, if XPath 3.1 is available, the lookup operator (?).

19.8.8.12 [Streamability of Variable References](#)

For variable references that are bound to the [streaming parameter](#) of a [declared-streamable stylesheet function](#), see the rules for the [streamability category](#) of the containing function, under [19.8.5 Classifying Stylesheet Functions](#).

In all other cases, variable references are [grounded](#) and [motionless](#).

19.8.8.13 [Streamability of the Context Item Expression](#)

The [posture](#) of the expression is the [context posture](#), and the [sweep](#) is [motionless](#).

Note:

Although `.` is intrinsically motionless, when used in certain contexts (such as `data(.`) the containing expression will be [consuming](#). This arises because of the [operand usage](#): the argument to `data`^{FO30} has usage [absorption](#), and the combination of a [motionless](#) operand with usage [absorption](#) leads to the containing expression being [consuming](#).

Similarly, if `.` is used where the [operand usage](#) is [navigation](#), the containing expression will be [free-ranging](#).

19.8.8.14 Streamability of Static Function Calls**Note:**

This section applies to static function calls written using the traditional syntax `F(X, Y, Z)` and equally to those using the new XPath 3.1 syntax `X => F(Y, Z)`

For calls to built-in functions, see [19.8.9 Classifying Calls to Built-In Functions](#).

For calls to [stylesheet functions](#), see [19.8.5 Classifying Stylesheet Functions](#).

For partial function applications (where one or more of the arguments is supplied as a `?` placeholder), see the rules at the end of this section.

For a call to a constructor function, the [19.8.1 General Rules for Streamability](#) apply. There is a single operand role (the argument to the function), with [operand usage absorption](#).

For a call to an [extension function](#), the [posture](#) and [sweep](#) are [implementation-defined](#).

If the function call is a partial function application (that is, if one or more of the arguments is given as a `?` placeholder), then:

1. If the function is focus-dependent and the [context posture](#) is not [grounded](#), then the function call is [roaming](#) and [free-ranging](#).
2. If the target of the function call is a [stylesheet function](#) that is [declared-streamable](#), and if the first argument is actually supplied (that is, this argument is not supplied as a `?` placeholder), and if the expression that is supplied as the first argument is not [grounded](#), then the function call is [roaming](#) and [free-ranging](#).
3. If the target is an [extension function](#), the [posture](#) and [sweep](#) are [implementation-defined](#).
4. Otherwise, the [general streamability rules](#) apply. The operands of a partial function application are the expressions actually supplied as arguments to the function, ignoring `?` place-holders; the corresponding [operand usage](#) is the [type-determined usage](#) based on the declared type of that argument.

19.8.8.15 Streamability of Named Function References

Let F be the function to which the `NamedFunctionRef` refers.

If F is focus-dependent and the [context posture](#) is not [grounded](#), then the [NamedFunctionRef](#) is [roaming](#) and [free-ranging](#).

If F is an [extension function](#), the [posture](#) and [sweep](#) are [implementation-defined](#).

Otherwise, the [NamedFunctionRef](#) is [grounded](#) and [motionless](#).

Note:

The main intent behind these rules is to ensure that the function item returned by a named function reference does not encapsulate a reference to a streamed node.

In the case of an expression such as `local-name#0`, implementations might be able to do better by pre-evaluating the function at the point where the named function reference occurs.

In the case of extension functions, implementations may be able to distinguish whether the function is focus-dependent, and decide the streamability of the named function reference accordingly.

19.8.8.16 [Streamability of Inline Function Declarations](#)

An inline function declaration that textually contains a variable reference bound to a [streaming parameter](#) (of some containing stylesheet function) is [roaming](#) and [free-ranging](#).

All other inline function declarations are [grounded](#) and [motionless](#).

Note:

It is not possible to pass a streamed node as an argument to a call to an inline function unless the declared type of the corresponding function parameter causes the node to be atomized: see [19.8.8.11 Streamability of Dynamic Function Calls](#). The only other way an inline function could access a streamed node is by having the streamed node in its closure, and this is prevented by the rule above.

19.8.8.17 [Streamability of Map Constructors](#)

The [posture](#) and [sweep](#) of a map constructor (see [21.4 Map Constructors](#)) are the same as the [posture](#) and [sweep](#) of the equivalent [xsl:map](#) instruction. The equivalent [xsl:map](#) instruction is formed by creating a sequence of [xsl:map-entry](#) instructions, one for each key/value pair in the map expression, where the key expression becomes the value of `xsl:map-entry/@key`, and the value expression becomes the value of `xsl:map-entry/@select`; this sequence of [xsl:map-entry](#) instructions is then wrapped in an [xsl:map](#) parent instruction.

For example, the map constructor `map{ 'red':false(), 'green':true() }` translates to the instruction:

```
<xsl:map>
  <xsl:map-entry key="'red'" select="false()" />
  <xsl:map-entry key="'green'" select="true()" />
</xsl:map>
```

The rules for the streamability of `xsl:map` appear in [19.8.4.23 Streamability of xsl:map](#).

See also [21.6 Maps and Streaming](#).

[19.8.8.18 Streamability of Lookup Expressions](#)

Lookup expressions for maps are defined in [21.5 The Map Lookup Operator](#), and are available in XSLT 3.0 whether or not XPath 3.1 is supported. Lookup expressions for arrays are defined in the XPath 3.1 specification (see [Section 3.11.3 The Lookup Operator \("?"\) for Maps and Arrays](#)^{XP31}), and are available only in XSLT 3.0 processors that provide the XPath 3.1 Feature (see [27.7 XPath 3.1 Feature](#)).

For the unary lookup operator, the `posture` and `sweep` of the expression `?X` are defined to be the same as the `posture` and `sweep` of the postfix lookup expression `.?X`.

For the postfix lookup expression `E?K`, the [general streamability rules](#) apply as follows:

1. In the wildcard form of the expression, `E?*`, there is only one operand, `E`. This has [operand usage inspection](#).
2. Where the construct `K` is an NCName, the expression `E?NAME` is treated as equivalent to `E?("NAME")`.
3. Where the construct `K` is an integer, the expression `E?N` is treated as equivalent to `E?(N)`.
4. In the general case where `K` is a parenthesized expression, the lookup expression `E?(K)` has two operands. The first operand `E` has [operand usage inspection](#), while the second operand `K` has [operand usage absorption](#).

[19.8.9 Classifying Calls to Built-In Functions](#)

This section describes the rules that determine the streamability of calls to built-in functions. These differ from user-written functions because it is known (defined in the specification) how nodes supplied as operands are used. Knowledge of the usage of each operand, together with the `posture` of the actual operands, is in most cases enough to determine the `posture` and `sweep` of the function result.

All the built-in functions are listed below. For most functions, a simple proforma is shown that indicates the operand usage of each argument, using the code (`A = absorption`, `I = inspection`, `T = transmission`, `N = navigation`). So, for example, the entry `fn:remove(T, A)` means that for the function `fn:remove#2`, the [operand usage](#) of the first argument is [transmission](#), and the [operand usage](#) of the second argument is [absorption](#). By reference to the general rules in [19.8.1 General Rules for Streamability](#), this demonstrates that if the [context posture](#) is [striding](#), the posture and sweep of the expression `sum(remove(*, 1))` will be [grounded](#) and [consuming](#) respectively.

For functions that default one of their arguments (typically to the context item), the relevant entry shows the equivalence, and the posture and sweep can in these cases be computed by filling in the default value for the relevant argument.

Some functions do not follow the general rules, and these are listed with a link to the section where the particular rules for that function are described.

- `array:append(I, N)`
- `array:filter(I, I)`
- `array:flatten(A)`
- `array:fold-left(I, N, I)`

- `array:fold-right(I, N, I)`
- `array:for-each(I, I)`
- `array:for-each-pair(I, I, I)`
- `array:get(I, A)`
- `array:head(I)`
- `array:insert-before(I, A, N)`
- `array:join(I)`
- `array:put(I, I, N)`
- `array:remove(I, A)`
- `array:reverse(I)`
- `array:size(I)`
- `array:sort(I)`
- `array:sort(I, A)`
- `array:sort(I, A, I)`
- `array:subarray(I, A)`
- `array:subarray(I, A, A)`
- `array:tail(I)`
- `fn:abs(A)`
- `fn:accumulator-after` – See [19.8.9.1 Streamability of the accumulator-after Function](#)
- `fn:accumulator-before` – See [19.8.9.2 Streamability of the accumulator-before Function](#)
- `fn:adjust-date-to-timezone(A)`
- `fn:adjust-date-to-timezone(A, A)`
- `fn:adjust-datetime-to-timezone(A)`
- `fn:adjust-datetime-to-timezone(A, A)`
- `fn:adjust-time-to-timezone(A)`
- `fn:adjust-time-to-timezone(A, A)`
- `fn:analyze-string(A, A)`
- `fn:analyze-string(A, A, A)`
- `fn:apply(A, I)`
- `fn:available-environment-variables()`
- `fn:available-system-properties()`
- `fn:avg(A)`
- `fn:base-uri()` – Equivalent to `fn:base-uri(.)`
- `fn:base-uri(I)`
- `fn:boolean(I)`
- `fn:ceiling(A)`

- `fn:codepoint-equal(A, A)`
- `fn:codepoints-to-string(A)`
- `fn:collation-key(A)`
- `fn:collation-key(A, A)`
- `fn:collection()`
- `fn:collection(A)`
- `fn:compare(A, A)`
- `fn:compare(A, A, A)`
- `fn:concat(A, A, A)`
- `fn:contains(A, A)`
- `fn:contains(A, A, A)`
- `fn:contains-token(A, A)`
- `fn:contains-token(A, A, A)`
- `fn:copy-of() – Equivalent to fn:copy-of(.)`
- `fn:copy-of(A)`
- `fn:count(I)`
- `fn:current – See 19.8.9.3 Streamability of the current Function`
- `fn:current-date()`
- `fn:current-dateTime()`
- `fn:current-group – See 19.8.9.4 Streamability of the current-group Function`
- `fn:current-grouping-key – See 19.8.9.5 Streamability of the current-grouping-key Function`
- `fn:current-merge-group – See 19.8.9.6 Streamability of the current-merge-group Function`
- `fn:current-merge-key – See 19.8.9.7 Streamability of the current-merge-key Function`
- `fn:current-output-uri()`
- `fn:current-time()`
- `fn:data() – Equivalent to fn:data(.)`
- `fn:data(A)`
- `fn:dateTime(A, A)`
- `fn:day-from-date(A)`
- `fn:day-from-dateTime(A)`
- `fn:days-from-duration(A)`
- `fn:deep-equal(A, A)`
- `fn:deep-equal(A, A, A)`
- `fn:deep-equal(A, A)`
- `fn:deep-equal(A, A, A)`
- `fn:default-collation()`

- `fn:default-language()`
- `fn:distinct-values(A)`
- `fn:distinct-values(A, A)`
- `fn:doc(A)`
- `fn:doc-available(A)`
- `fn:document(A)`
- `fn:document(A, I)`
- `fn:document-uri()` – Equivalent to `fn:document-uri(.)`
- `fn:document-uri(I)`
- `fn:element-available(A)`
- `fn:element-with-id(x)` – Equivalent to `fn:element-with-id(x, .)`
- `fn:element-with-id(A, N)`
- `fn:empty(I)`
- `fn:encode-for-uri(A)`
- `fn:ends-with(A, A)`
- `fn:ends-with(A, A, A)`
- `fn:environment-variable(A)`
- `fn:error()` – Equivalent to `fn:error(x, x, x, x, x, .)`
- `fn:error(x)` – Equivalent to `fn:error(x, x, x, x, .)`
- `fn:error(x, x)` – Equivalent to `fn:error(x, x, .)`
- `fn:error(A, A, N)`
- `fn:escape-html-uri(A)`
- `fn:exactly-one(T)`
- `fn:exists(I)`
- `fn:false()`
- `fn:filter(N, I)`
- `fn:floor(A)`
- `fn:fold-left(N, A, I)`
- `fn:fold-right` – See [19.8.9.9 Streamability of the fold-right Function](#)
- `fn:for-each(N, I)`
- `fn:for-each-pair(N, N, I)`
- `fn:format-date(A, A)`
- `fn:format-date(A, A, A, A, A)`
- `fn:format-dateTime(A, A)`
- `fn:format-dateTime(A, A, A, A, A)`
- `fn:format-integer(A, A)`

- `fn:format-integer(A, A, A)`
- `fn:format-number(A, A)`
- `fn:format-number(A, A, A)`
- `fn:format-time(A, A)`
- `fn:format-time(A, A, A, A, A)`
- `fn:function-arity(A)`
- `fn:function-available(A)`
- `fn:function-available(A, A)`
- `fn:function-lookup` – See [19.8.9.12 Streamability of the function-lookup Function](#)
- `fn:function-name(A)`
- `fn:generate-id()` – Equivalent to `fn:generate-id(.)`
- `fn:generate-id(I)`
- `fn:has-children()` – Equivalent to `fn:has-children(.)`
- `fn:has-children(I)`
- `fn:head(T)`
- `fn:hours-from-dateTime(A)`
- `fn:hours-from-duration(A)`
- `fn:hours-from-time(A)`
- `fn:id(x)` – Equivalent to `fn:id(x, .)`
- `fn:id(A, N)`
- `fn:idref(x)` – Equivalent to `fn:idref(x, .)`
- `fn:idref(A, N)`
- `fn:implicit-timezone()`
- `fn:in-scope-prefixes(I)`
- `fn:index-of(A, A)`
- `fn:index-of(A, A, A)`
- `fn:innermost` – See [19.8.9.13 Streamability of the innermost Function](#)
- `fn:insert-before(T, A, T)`
- `fn:iri-to-uri(A)`
- `fn:json-doc(A)`
- `fn:json-doc(A, I)`
- `fn:json-to-xml(A)`
- `fn:json-to-xml(A, I)`
- `fn:key(x, x)` – Equivalent to `fn:key(x, x, /)`
- `fn:key(A, A, N)`
- `fn:lang(x)` – Equivalent to `fn:lang(x, .)`

- `fn:lang(A, I)`
- `fn:last` – See [19.8.9.14 Streamability of the last Function](#)
- `fn:load-xquery-module(A)`
- `fn:load-xquery-module(A, I)`
- `fn:local-name()` – Equivalent to `fn:local-name(.)`
- `fn:local-name(I)`
- `fn:local-name-from-QName(A)`
- `fn:lower-case(A)`
- `fn:matches(A, A)`
- `fn:matches(A, A, A)`
- `fn:max(A)`
- `fn:max(A, A)`
- `fn:min(A)`
- `fn:min(A, A)`
- `fn:minutes-from-dateTime(A)`
- `fn:minutes-from-duration(A)`
- `fn:minutes-from-time(A)`
- `fn:month-from-date(A)`
- `fn:month-from-dateTime(A)`
- `fn:months-from-duration(A)`
- `fn:name()` – Equivalent to `fn:name(.)`
- `fn:name(I)`
- `fn:namespace-uri()` – Equivalent to `fn:namespace-uri(.)`
- `fn:namespace-uri(I)`
- `fn:namespace-uri-for-prefix(A, I)`
- `fn:namespace-uri-from-QName(A)`
- `fn:nilled()` – Equivalent to `fn:nilled(.)`
- `fn:nilled(I)`
- `fn:node-name()` – Equivalent to `fn:node-name(.)`
- `fn:node-name(I)`
- `fn:normalize-space()`
- `fn:normalize-space(A)`
- `fn:normalize-unicode(A)`
- `fn:normalize-unicode(A, A)`
- `fn:not(I)`
- `fn:number()` – Equivalent to `fn:number(.)`

- `fn:number(A)`
- `fn:one-or-more(T)`
- `fn:outermost` – See [19.8.9.15 Streamability of the outermost Function](#)
- `fn:parse-ietf-date(A)`
- `fn:parse-json(A)`
- `fn:parse-json(A, I)`
- `fn:parse-xml(A)`
- `fn:parse-xml-fragment(A)`
- `fn:path()` – Equivalent to `fn:path(.)`
- `fn:path(N)`
- `fn:position` – See [19.8.9.16 Streamability of the position Function](#)
- `fn:prefix-from-QName(A)`
- `fn:QName(A, A)`
- `fn:random-number-generator()`
- `fn:random-number-generator(A)`
- `fn:regex-group(A)`
- `fn:remove(T, A)`
- `fn:replace(A, A, A)`
- `fn:replace(A, A, A, A)`
- `fn:resolve-QName(A, I)`
- `fn:resolve-uri(A)`
- `fn:resolve-uri(A, A)`
- `fn:reverse` – See [19.8.9.17 Streamability of the reverse Function](#)
- `fn:root` – See [19.8.9.18 Streamability of the root Function](#)
- `fn:round(A)`
- `fn:round(A, A)`
- `fn:round-half-to-even(A)`
- `fn:round-half-to-even(A, A)`
- `fn:seconds-from-datetime(A)`
- `fn:seconds-from-duration(A)`
- `fn:seconds-from-time(A)`
- `fn:serialize(A)`
- `fn:serialize(A, A)`
- `fn:snapshot()` – Equivalent to `fn:snapshot(.)`
- `fn:snapshot(A)`
- `fn:sort(N)`

- `fn:sort(N, A)`
- `fn:sort(N, A, I)`
- `fn:starts-with(A, A)`
- `fn:starts-with(A, A, A)`
- `fn:static-base-uri()`
- `fn:stream-available(A)`
- `fn:string() – Equivalent to fn:string(.)`
- `fn:string(A)`
- `fn:string-join(A)`
- `fn:string-join(A, A)`
- `fn:string-length()`
- `fn:string-length(A)`
- `fn:string-to-codepoints(A)`
- `fn:subsequence(T, A)`
- `fn:subsequence(T, A, A)`
- `fn:substring(A, A)`
- `fn:substring(A, A, A)`
- `fn:substring-after(A, A)`
- `fn:substring-after(A, A, A)`
- `fn:substring-before(A, A)`
- `fn:substring-before(A, A, A)`
- `fn:sum(A)`
- `fn:sum(A, A)`
- `fn:system-property(A)`
- `fn:tail(T)`
- `fn:timezone-from-date(A)`
- `fn:timezone-from-datetime(A)`
- `fn:timezone-from-time(A)`
- `fn:tokenize(A)`
- `fn:tokenize(A, A)`
- `fn:tokenize(A, A, A)`
- `fn:trace(A)`
- `fn:trace(T, A)`
- `fn:transform(I)`
- `fn:translate(A, A, A)`
- `fn:true()`

- `fn:type-available(A)`
- `fn:unordered(T)`
- `fn:unparsed-entity-public-id(x)` – Equivalent to `fn:unparsed-entity-public-id(x, /)`
- `fn:unparsed-entity-public-id(A, I)`
- `fn:unparsed-entity-uri(x)` – Equivalent to `fn:unparsed-entity-uri(x, /)`
- `fn:unparsed-entity-uri(A, I)`
- `fn:unparsed-text(A)`
- `fn:unparsed-text(A, A)`
- `fn:unparsed-text-available(A)`
- `fn:unparsed-text-available(A, A)`
- `fn:unparsed-text-lines(A)`
- `fn:unparsed-text-lines(A, A)`
- `fn:upper-case(A)`
- `fn:uri-collection()`
- `fn:uri-collection(A)`
- `fn:xml-to-json(A)`
- `fn:xml-to-json(A, I)`
- `fn:year-from-date(A)`
- `fn:year-from-datetime(A)`
- `fn:years-from-duration(A)`
- `fn:zero-or-one(T)`
- `map:contains(I, A)`
- `map:entry(A, N)`
- `map:find(I, A)`
- `map:for-each(I, I)`
- `map:get(I, A)`
- `map:keys(I)`
- `map:merge(I)`
- `map:merge(I, I)`
- `map:put(I, A, N)`
- `map:remove(I, A)`
- `map:size(I)`
- `math:acos(A)`
- `math:asin(A)`
- `math:atan(A)`
- `math:atan2(A, A)`

- `math:cos(A)`
- `math:exp(A)`
- `math:exp10(A)`
- `math:log(A)`
- `math:log10(A)`
- `math:pi()`
- `math:pow(A, A)`
- `math:sin(A)`
- `math:sqrt(A)`
- `math:tan(A)`

[19.8.9.1 Streamability of the accumulator-after Function](#)

See also [18.2.9 Streamability of Accumulators](#).

The `posture` of the function call is in all cases `grounded`.

The `sweep` is determined by applying the following rules, in order:

1. If the first argument (the accumulator name) is not `motionless`, the function is `free-ranging`.
2. If the `context posture` is `grounded`, the function is `motionless`.
3. If the `context item type` has an empty intersection with $U\{document-node(), element()\}$ (that is, if the context item cannot have children), the function is `motionless`.
4. If the function call is contained in the `select` expression or contained sequence constructor of an `xsl:accumulator-rule` specifying `phase="start"`, then it is `free-ranging`.
5. If the function call is contained in the `select` expression or contained sequence constructor of an `xsl:accumulator-rule` specifying `phase="end"`, then it is `motionless`.
6. If no enclosing node of the function call is part of a `sequence constructor`, then it is `free-ranging`. For this purpose, the **enclosing nodes** of a function call are the attribute or text node that immediately contains the XPath expression in which the function call appears, and its ancestors.
7. If the `focus-setting container` of the function call is different from the `focus-setting container` of the innermost containing `instruction`, then the function is `free-ranging`.
8. If no enclosing node N of the function call has a preceding sibling node P such that (a) N and P are part of the same `sequence constructor`, and (b) the `sweep` of P is `consuming`, then the function call is `consuming`. (The term **enclosing node** is defined above.)
9. Otherwise, the function call is `motionless`.

Note:

The following notes apply to the above rules with matching numbers:

1. This rule prevents the accumulator name being computed by reading the streamed source document. This is disallowed primarily because there is no conceivable use case for doing it.
2. If the context posture is grounded, then the target of the accumulator is not a streamed node, so no streaming restrictions apply.
3. If the context item is a childless node (such as a text node), then both the pre-descent and post-descent values of the accumulator can be computed before evaluating any user-written constructs that access this node; there are therefore no constraints on where a call to [accumulator-after](#) can appear.
4. This rule ensures that when computing the pre-descent value of an accumulator for a particular streamed node, the post-descent values of accumulators for that node are not available.
5. This rule states that the post-descent value of an accumulator is allowed to depend on the post-descent values of other accumulators for the same node. There is a rule preventing cycles [see [ERR_XTDE3400](#)].
6. This rule prevents the use of the function (when applied to a streamed node) in contexts like the `use` attribute of [xsl:key](#). It allows its use in the attributes of an [instruction](#) or [literal result element](#), or in a [text value template](#). It does not allow use in an [xsl:sort](#) or [xsl:param](#) element, as these elements do not form part of a sequence constructor (see [5.7 Sequence Constructors](#)).
7. This rule prevents the use of the function (when applied to a streamed node) in contexts such as predicates, or the right-hand side of the / operator. The focus for evaluation of the function must be the same as the focus for a containing sequence constructor. Sequence constructors are treated differently from all other constructs for this purpose in that their operands (the contained instructions) are treated as ordered: in conjunction with the next rule, this rule is assuming that instructions in a sequence constructor that follow a [consuming](#) instruction are evaluated after the [consuming](#) instruction and therefore have access to the post-descent accumulator value.
8. This rule is subtle, and has a number of consequences. In these notes, the term **instruction** should be read as including all nodes making up a sequence constructor, including XSLT instructions, extension instructions, literal result elements, and text nodes containing text value templates.
 - In a sequence constructor that contains a [consuming](#) instruction such as `<xsl:apply-templates/>`, it allows any number of calls on [accumulator-after](#) to appear in instructions that follow the call on `<xsl:apply-templates/>`.
 - In such a sequence constructor it prevents a call on [accumulator-after](#) from appearing in an instruction that precedes the `<xsl:apply-templates/>`, because there would then be two [consuming](#) instructions.
 - In a sequence constructor that contains calls on [accumulator-after](#), and contains no other [consuming](#) construct, the first instruction that contains a call on [accumulator-after](#) is consuming (unless it contains more than one such call, in which case it is free-ranging), and subsequent instructions containing such a call are motionless. So it is possible to have two or more calls on [accumulator-after](#) provided they appear in different instructions, which allows the analysis to assume an order of execution.
 - It prevents a call on [accumulator-after](#) from appearing in the same instruction as another consuming construct: for example it disallows `concat(child::p, accumulator-after('a'))`. This rule preserves the ability to evaluate the arguments of the `concat` function in any order.

- It disallows a call on `accumulator-after` from appearing in a sequence constructor that is required to be motionless, for example within `xsl:sort`.
 - The reference to a “preceding sibling node within the same sequence constructor” is carefully worded to ensure that preceding siblings among the children of `xsl:fork` are not taken into account; the children of `xsl:fork` are sibling instructions, but do not constitute a sequence constructor. The term also excludes elements such as `xsl:param` and `xsl:sort` that may precede a sequence constructor but are not part of it.
9. The final rule states that if none of the previous rules apply, the function is considered motionless. This applies when the `accumulator-after` appears after a consuming instruction within the same sequence constructor.
- Note also that a call to `accumulator-after` can safely appear within a construct such as a named template or (non-streamable) stylesheet function; this is safe because the rules ensure that in such situations, the context item cannot be a streamed node.

Dynamic invocation of `accumulator-after` is covered by the rules in [10.3.6 Dynamic Access to Functions](#). These rules ensure that a function item cannot include a streamed node in its closure; circumventing the streamability rules for `accumulator-after` by making a dynamic call is therefore not possible.

[19.8.9.2 Streamability of the accumulator-before Function](#)

See also [18.2.9 Streamability of Accumulators](#).

The `posture` and `sweep` of the function call are assessed as follows:

1. If the argument to `accumulator-before` is motionless, the function call is `grounded` and `motionless`.
2. Otherwise, the function call is `roaming` and `free-ranging`.

[19.8.9.3 Streamability of the current Function](#)

The `sweep` and `posture` of a call to the `current` function are determined as follows:

1. If the call appears within a pattern, then climbing and motionless.
- Note:**
- The call to `current` will always be within a predicate of the pattern. The use of climbing posture here allows predicates such as `[@class = current()]/@class`, while disallowing downwards navigation from the node returned by the function.
2. Otherwise, let E be the outermost containing XPath expression of the call to the `current` function.
 3. If the `context posture` of E is `grounded`, then `motionless` and `grounded`.
 4. If the path in the expression tree that connects the call on `current` to E (excluding E itself) contains an expression that is a `higher-order operand` of its parent expression, then `motionless` and `climbing`.

Note:

Many common uses of the `current`, such as `//p[@class=current()]/@class`, fall into this category: a predicate is a higher-order operand of its containing filter expression.

The use of `climbing` posture here might seem unrelated to its usual connection with the ancestor axis. The explanation (apart from the fact that it happens to produce the right results) lies in the fact that at the point where the `current` call is evaluated, the node it returns will always be an ancestor-or-self of the context node, as a consequence of the fact that the containing XPath expression is required to be either `motionless` or `consuming`.

The effect of the rule is to allow expressions such as `//*[name() = name(current())]` or `//*[@ref = current()]/@id`.

5. Otherwise, the `posture` is the `context posture`, and the `sweep` is `motionless`.

19.8.9.4 Streamability of the current-group Function

The `sweep` and `posture` of a call C to the `current-group` function are as follows:

1. If all the following conditions are true:

- a. C has a containing `xsl:for-each-group` instruction (call it F)
- b. The path in the construct tree that connects C to the sequence constructor forming the body of F is such that no child construct is a `higher-order operand` of its parent
- c. The `focus-setting container` of C is F

then the `sweep` and `posture` of C are the `sweep` and `posture` of the `select` expression of F .

2. Otherwise, `roaming` and `free-ranging`.

Note:

Informally, for streamed evaluation to be possible, a call to `current-group` must not appear in a construct that is evaluated repeatedly. For example, the expression `for $i in 1 to 10 return current-group()` would not be streamable.

19.8.9.5 Streamability of the current-grouping-key Function

A call to the `current-grouping-key` function is grounded and motionless.

19.8.9.6 Streamability of the current-merge-group Function

A call to the `current-merge-group` function is `grounded` and `motionless`.

Note:

This is because the nodes to be merged are always snapshots, and therefore [grounded](#): see [15.4 Streamable Merging](#).

19.8.9.7 Streamability of the current-merge-key Function

A call to the [current-merge-key](#) function is [grounded](#) and [motionless](#).

19.8.9.8 Streamability of the fold-left^{FO30} Function

The function call `fold-left($seq, $zero, $f)`, follows the [general streamability rules](#), with the first argument `$seq` having [type-determined usage](#) based on the type of the second argument of the function supplied as `$f`.

For example, given the call `fold-left(*transaction, 0, function($x as xs:decimal, $y as xs:decimal) as xs:decimal {$x+$y})`, the [operand usage](#) of the argument `/*transaction` is determined by the declared type of `$y`, namely `xs:decimal`. Since this is an atomic type, the [type-determined usage](#) is [absorption](#). Applying this to the general streamability rules, the function call is [grounded](#) and [consuming](#).

19.8.9.9 Streamability of the fold-right^{FO30} Function

The function follows the [general streamability rules](#), with the first argument having [operand usage navigation](#) to reflect the fact that the supplied sequence is processed in reverse order.

Note:

The same considerations apply as for the [reverse^{FO30}](#) function: see [19.8.9.17 Streamability of the reverse Function](#).

19.8.9.10 Streamability of the for-each^{FO30} Function

The function call `for-each($seq, $f)`, follows the [general streamability rules](#), with the first argument `$seq` having [type-determined usage](#) based on the type of the (single) argument of the function supplied as `$f`.

For example, given the call `for-each(*transaction, function($x as xs:decimal) as xs:decimal {abs($x)})`, the [operand usage](#) of the argument `/*transaction` is determined by the declared type of `$x`, namely `xs:decimal`. Since this is an atomic type, the [type-determined usage](#) is [absorption](#). Applying this to the general streamability rules, the function call is [grounded](#) and [consuming](#).

Note:

In practice, the `filter`^{FO30} function is streamable if either (a) the supplied sequence is grounded, or (b) the supplied function is statically known to atomize its argument.

19.8.9.11 Streamability of the `for-each-pair`^{FO30} Function

The function call `for-each($seq1, $seq2, $f)`, follows the [general streamability rules](#), where:

1. The first argument `$seq1` has [type-determined usage](#) based on the type of the first argument of the function supplied as `$f`.
2. The second argument `$seq2` has [type-determined usage](#) based on the type of the second argument of the function supplied as `$f`

Note:

In practice, the `for-each-pair`^{FO30} function is streamable provided (a) at most one of the input sequences is consuming, and (b) either (i) that input sequence is grounded, or (ii) the supplied function is statically known to atomize the relevant argument.

If it is necessary to combine two sequences that are both streamed, consider using `xsl:merge`.

19.8.9.12 Streamability of the `function-lookup`^{FO30} Function

See [10.3.6 Dynamic Access to Functions](#) for special rules that relate to streamability of calls to the `function-lookup`^{FO30} function.

With the caveats given there, the function follows the [general streamability rules](#), for a function with two arguments that both have [operand usage absorption](#).

19.8.9.13 Streamability of the `innermost`^{FO30} Function

The function follows the [general streamability rules](#), with the first argument having [operand usage navigation](#). This is to reflect the fact that the processing is not strictly sequential: it cannot be determined that a node is part of the result sequence of `innermost`^{FO30} until all its descendants have been read.

19.8.9.14 Streamability of the `last`^{FO30} Function

If the [context posture](#) for a call on the `last`^{FO30} function is [striding](#), [crawling](#), or [roaming](#), then the [posture](#) of the function is [roaming](#), and the [sweep](#) is [free-ranging](#).

In all other cases the function is [grounded](#) and [motionless](#).

Note:

The cases where `last`^{FO30} can be used without affecting streamability are where the context item is either grounded or climbing. The latter condition makes expressions like `ancestor::*[@xml:space][last()]` streamable.

There are special rules restricting the use of `last`^{FO30} in the predicate of a pattern: see [19.8.10 Classifying Patterns](#).

Note that there are no restrictions preventing the use of `last()` when the context posture is grounded. The implications of this are discussed in [19.7 Grounded Consuming Constructs](#). In the case where the sequence being processed is delivered by a consuming expression, using `last()` may result in this sequence being buffered in memory.

[**19.8.9.15 Streamability of the outermost**](#)^{FO30} [**Function**](#)

The single argument to this function has [operand usage transmission](#).

The streamability of the function call follows the [general streamability rules](#) with one exception: if the posture of the argument is crawling, then the posture of the result is striding.

Note:

There are cases where the streaming rules allow the construct `outermost(//para)` but do not allow `//para`; the function can therefore be useful in cases where it is known that `para` elements will not be nested, as well as cases where the application actually wishes to process all `para` elements except those that are nested within another.

By contrast, the `innermost`^{FO30} function offers no streaming benefits. Although it delivers a subset of the input nodes as its result, in the correct order, it is classed as navigational because it needs to look ahead in the input stream before deciding whether a node can be included in the result.

[**19.8.9.16 Streamability of the position**](#)^{FO30} [**Function**](#)

The `position`^{FO30} function follows the [general streamability rules](#). Since it has no operands, this means it is grounded and motionless.

Note:

Within an expression, there are no special difficulties in evaluating the `position`^{FO30} function.

It does have special treatment within a predicate of a pattern, however: a pattern is not motionless if it contains a call to `position`^{FO30}, as explained in [19.8.10 Classifying Patterns](#).

19.8.9.17 Streamability of the reverse^{FO30} Function

The `reverse`^{FO30} function follows the [general streamability rules](#), with its operand classified as having [operand usage navigation](#).

Note:

This means in effect that a call on `reverse`^{FO30} is not streamable unless the operand is grounded. This may cause few surprises:

- The expression `reverse(/emp/copy-of())` is considered streamable, although all the `emp` elements will typically need to be in memory at the same time. The explanation here is that the streamability rules do not attempt to restrict the amount of memory used for data that is explicitly copied by use of a function such as `copy-of`.
- The expression `reverse(ancestor::* / name())` is considered non-streamable, because the operand is not grounded. This problem can be circumvented by rewriting the expression as `reverse(ancestor::*/name())`

19.8.9.18 Streamability of the root^{FO30} Function

The zero-argument function `root()` is equivalent to `root(.)`.

Given the expression `root(X)`, if the [static type](#) of `X` is $U\{document-node()\}$, and if its [posture](#) is [striding](#), then `root(X)` is rewritten as `X`. Otherwise, it is rewritten as `head((X)/ancestor-or-self::node())`.

Streamability analysis is then applied to the rewritten expression.

Note:

Because path expressions starting with `/` are rewritten to use the `root`^{FO30} function, this ensures that a leading slash is ignored if the context item is a document node, for example within a template rule with `match="/".` This improves streamability, because upwards navigation followed by downward navigation is disallowed.

19.8.10 Classifying Patterns

Note:

Patterns differ from other kinds of construct in that they are not composable in the same way. It is best to think of a pattern as specialized syntax for a function that takes an item as its argument and returns a boolean: true if the pattern matches the item, otherwise false. The [static type](#) of a pattern is therefore taken as $U\{xs:boolean\}$ (this is not to be confused with the type of the items that the pattern is capable of matching).

The sweep of a pattern is either motionless or free-ranging. (Although there are patterns that could in principle be evaluated by consuming the element node that they match, these are of no interest in the analysis, so they are classified as free-ranging.)

The posture of a pattern is grounded if the pattern is motionless, or roaming otherwise. (This reflects the fact that a pattern always returns a boolean result; it never returns a node in a streamed document.)

Informally, a motionless pattern is one that can be evaluated by a streaming processor when the input stream is positioned at the start of the node being matched, without advancing the input stream.

A pattern is motionless if and only if it satisfies all the following conditions:

1. The pattern does not contain a RootedPath.
2. If the pattern contains predicates, then every top-level Predicate in the pattern satisfies all the following conditions:
 - a. The expression immediately contained in the predicate is motionless, when assessed with a context posture of striding, and a context item type set to the static type of the expression to which the predicate applies, determined using the rules in [19.1 Determining the Static Type of a Construct](#).
 - b. The predicate is a non-positional predicate.

The use of the term **top-level** in this rule means that predicates that are nested within other predicates do not themselves have to be non-positional, though they may play a role in the analysis of top-level predicates.

3. The pattern does not contain (at any depth) a variable reference that is bound to a streaming parameter. (See [19.8.8.14 Streamability of Static Function Calls](#)).

[DEFINITION: A predicate is a **non-positional predicate** if it satisfies both of the following conditions:

1. The predicate does not contain a function call or named function reference to any of the following functions, unless that call or reference occurs within a nested predicate:
 - a. position^{FO30}
 - b. last^{FO30}
 - c. function-lookup^{FO30}.

Note:

The exception for nested predicates is there to ensure that patterns such as `match="p[@code = $status[last()]]` are not disqualified.

2. The expression immediately contained in the predicate is a non-numeric expression. An expression is non-numeric if the intersection of its static type (see [19.1 Determining the Static Type of a Construct](#)) with $U\{xs:decimal, xs:double, xs:float\}$ is $U\{\}$.

]

Note:

A non-positional predicate can be evaluated by considering each item in the filtered sequence independently; the result never depends on the position of other items in the sequence or the length of the sequence.

A pattern that is not [motionless](#) is classified as [free-ranging](#).

The following list shows examples of motionless patterns:

- /
- *
- /*
- p
- p|q
- p/q
- p[@status='red']
- p[base-uri()]
- p[@class or @style]
- p[@status]
- p[@status = \$status-codes[1]]
- p[@class | @style]
- p[contains(@class, ':')]
- p[substring-after(@class, ':')]
- p[ancestor::*:xml:lang]]
- text()[starts-with(., '\$')]
- @price
- @price[starts-with(., '\$')]
- //p/text()[. = 'Introduction']
- document-node(element(html)) (Note: this is classified as motionless even though testing a document node against the pattern might require a small amount of look-ahead.)

The following list shows examples of patterns that are not motionless, explaining why not:

- id('abc') (contains a RootedPath)
- \$doc//p (contains a RootedPath)
- p[b] (the predicate is not motionless)
- p[. = 'Introduction'] (the predicate is not motionless)
- p[starts-with(., '\$')] (the predicate is not motionless)
- p[preceding-sibling::p[1] = ''] (the predicate is not motionless)

- $p[1]$ (contains a positional predicate: return type is numeric)
- $p[$pnum + 1]$ (contains a positional predicate: return type is numeric)
- $p[data(@status)]$ (contains a positional predicate: return type is potentially numeric)
- $p[position() gt 2]$ (contains a positional predicate: calls `position()`)
- $p[last()]$ (contains a positional predicate: calls `last()`)

19.9 Examples of Streamability Analysis

The examples in this section are intended to illustrate how the streamability rules are applied “top down” to establish whether template rules are guaranteed streamable.

Example: A recursive-descent template rule

Consider the following template rule, where mode s is defined with `streamable="yes"`:

```
<xsl:template match="para" mode="s">
  <div class="para">
    <xsl:apply-templates mode="s"/>
  </div>
</xsl:template>
```

The processor is required to establish that this template meets the streamability rules. Specifically, as stated in [6.6.4 Streamable Templates](#), it must satisfy three conditions:

1. The match pattern must be [motionless](#).
2. The body of the template rule must be [grounded](#).
3. The initializers of any template parameters must be [motionless](#).

The third condition is satisfied trivially because there are no parameters.

The first rule depends on the rules for assessing patterns, which are given in [19.8.10 Classifying Patterns](#). This pattern is motionless because (a) it does not contain a `RootedPath`, and (b) it contains no predicates.

So it remains to determine that the body of the template is [grounded](#). The proof of this is as follows:

1. The sequence constructor forming the body of the template is assessed according to the rules in [19.8.3 Classifying Sequence Constructors](#), which tell us that there is a single operand (the `<div>` [literal result element](#)) which has [operand usage](#) $U = \text{transmission}$.
2. The assessment of the sequence constructor uses the [general streamability rules](#). These rules require us to determine the type T , sweep S , posture P , and usage U of each operand. We have already established that there is a single operand, with $U = \text{transmission}$. Section [19.1 Determining the Static Type of a Construct](#) tells us that for all instructions, we can take $T = U\{*\}$. The [posture](#) P and [sweep](#) S of the literal result element are established as follows:
 - a. The rules for literal result elements (specifically the `<div>` element) are given in [19.8.4.1 Streamability of Literal Result Elements](#). This particular literal result element has only one operand (its contained sequence constructor), with [operand usage](#) $U = \text{absorption}$.
 - b. The [general streamability rules](#) again apply. Again the [static type](#) T of the operand is $U\{*\}$, and we need to determine the [posture](#) P and [sweep](#) S .
 - c. To determine the posture and sweep of this sequence constructor (the one that contains the `xsl:apply-templates` instruction) we refer again to the [general streamability rules](#).
 - i. The sequence constructor has a single operand (the `xsl:apply-templates` instruction); again $U = \text{transmission}$, $T = U\{*\}$.
 - ii. The [posture](#) P and [sweep](#) S of the `xsl:apply-templates` instruction are established as follows:
 - A. The rules that apply are in [19.8.4.5 Streamability of xsl:apply-templates](#).
 - B. Rule 1 does not apply because the `select` expression (which defaults to `child::node()`) is not [grounded](#). This is a consequence of the rules in [19.8.8.9 Streamability of Axis Steps](#), specifically:

- I. The context posture of the axis step is established by the template rule as a whole, as striding.
- II. Therefore rules 1 and 2 do not apply.
- III. The statically-inferred context item type is derived from the match pattern (`match="para"`). This gives a type of $U\{element()\}$. The child axis for element nodes is not necessarily empty, so rule 3 does not apply.
- IV. Rule 4 does not apply because there are no predicates.
- V. So the posture and sweep of the axis step `child::node()` are given by the table in rule 5. The entry for (context posture = striding, axis = child) gives a posture of striding and a sweep of consuming.
- VI. So the select expression is not grounded. (The same result can be reached intuitively: an expression that selects streamed nodes will never be grounded.)
- C. Rule 2 does not apply because there is no xsl:sort element.
- D. Rule 3 does not apply because the mode is declared with `streamable="yes"`.
- E. So the posture P and sweep S of the xsl:apply-templates instruction are established by the general streamability rules, as follows:
 - I. There is a single operand, the implicit `select="child::node()"` expression, with usage $U = \text{absorption}$.
 - II. We have already established that for this operand, the posture $P = \text{striding}$ and the sweep $S = \text{consuming}$.
 - III. By the rules in [19.1 Determining the Static Type of a Construct](#), the type T of the select expression is `node()`.
 - IV. In the general streamability rules, the adjusted sweep S' for an operand with ($P = \text{striding}$, $U = \text{absorption}$) is consuming,
 - V. Rule 2(d) then applies, so the xsl:apply-templates instruction is consuming and grounded.
- iii. So the sequence constructor that contains the xsl:apply-templates instruction has one operand with $U = \text{transmission}$, $T = \text{item}()$, $P = \text{grounded}$, $S = \text{consuming}$. Rule 2(d) of the general streamability rules applies, so the sequence constructor itself has $P = \text{grounded}$, $S = \text{consuming}$.
- d. So the literal result element has one operand with $U = \text{absorption}$, $T = \text{item}()$, $P = \text{grounded}$, $S = \text{consuming}$. Rule 2(d) of the general streamability rules applies, so the literal result element has $P = \text{grounded}$, $S = \text{consuming}$.
- 3. So the sequence constructor containing the literal result element has one operand with $U = \text{transmission}$, $T = \text{item}()$, $P = \text{grounded}$, $S = \text{consuming}$. Rule 2(d) of the general streamability rules applies, so this sequence constructor itself has $P = \text{grounded}$, $S = \text{consuming}$.
- 4. So we have established that the sequence constructor forming the body of the template rule is grounded.

Therefore, since the other conditions are also satisfied, the template is guaranteed-streamable.

The analysis presented above could have been simplified by taking into account the fact that the streamability properties of a sequence constructor containing a single instruction are identical to the properties of that instruction. This simplification will be exploited in the next example.

Example: An aggregating template rule

Consider the following template rule, where mode s is defined with `streamable="yes"`:

```
<xsl:template match="transactions[@currency='USD']" mode="s">
  <total><xsl:value-of select="sum(transaction/@value)" /></total>
</xsl:template>
```

Again, as stated in [6.6.4 Streamable Templates](#), it must satisfy three conditions:

1. The match pattern must be motionless.
2. The body of the template rule must be grounded.
3. The initializers of any template parameters must be motionless.

The third condition is satisfied trivially because there are no parameters.

The first rule depends on the rules for assessing patterns, which are given in [19.8.10 Classifying Patterns](#). This pattern is motionless because (a) it is not a `RootedPath`, and (b) every predicate is motionless and non-positional. The analysis that proves the predicate is motionless and non-positional proceeds as follows:

1. First establish that that the expression `@currency='USD'` is motionless, as follows:
 - a. The predicate is a general comparison (`GeneralComp`) which follows the [general streamability rules](#).
 - b. There are two operands: an `AxisStep` with a defaulted `ForwardAxis`, and a `Literal`. Both operand roles are absorption.
 - c. The left-hand operand has type $T = \text{attribute}()$. Its posture and sweep are determined by the rules in [19.8.8.9 Streamability of Axis Steps](#). The context posture is striding, so the posture and sweep are determined by the entry in the table (rule 5) with context posture = striding, axis = `attribute`: that is, the result posture is striding and the sweep is motionless.
 - d. The right-hand operand, being a literal, is grounded and motionless.
 - e. In the [general streamability rules](#), rule 2(e) applies, so the predicate is grounded and motionless
2. Now establish that that the expression `@currency='USD'` is non-positional, as follows:
 - a. Rule 1 is satisfied: the predicate does not call `position`^{FO30}, `last`^{FO30}, or `function-lookup`^{FO30}.
 - b. Rule 2 is satisfied: the expression `@currency='USD'` is non-numeric. The static type of the expression is determined using the rules in [19.1 Determining the Static Type of a Construct](#) as $U\{\text{xs:boolean}\}$, and this has no intersection with $U\{\text{xs:decimal}, \text{xs:double}, \text{xs:float}\}$.

So both conditions in [19.8.10 Classifying Patterns](#) are satisfied, and the pattern is therefore motionless.

It remains to show that the body of the template rule is grounded. The proof of this is as follows. Unlike the previous example, the analysis is shown in simplified form; in particular the two sequence constructors which each contain a single instruction are ignored, and replaced in the construct tree by their contained instruction.

1. We need to show that the `<total>` literal result element is grounded.
2. The rules that apply are in [19.8.4.1 Streamability of Literal Result Elements](#).
3. These rules refer to the [general streamability rules](#). There is one operand, the `xsl:value-of` child element, which has operand usage $U = \text{absorption}$, and type $T = \text{item}()$.

4. So we need to determine the posture and sweep of the xsl:value-of instruction.
 - a. The rules are given in [19.8.4.40 Streamability of xsl:value-of](#).
 - b. The general streamability rules apply. There is one operand, the expression `sum(transaction/@value)`, which has operand usage $U = \text{absorption}$.
 - c. The type T of this operand is the return type defined in the signature of the sum^{FO30} function, that is, `xs:anyAtomicType`.
 - d. The posture P and sweep S are established as follows:
 - i. The rules that apply to the call on sum^{FO30} are given in [19.8.9 Classifying Calls to Built-In Functions](#).
 - ii. The relevant proforma is `fn:sum(A)`, indicating that the general streamability rules apply, and that there is a single operand with usage $U = \text{absorption}$.
 - iii. The type T of the operand `transaction/@value` is determined (by the rules in [19.1 Determining the Static Type of a Construct](#)) as `attribute()`.
 - iv. The posture P and sweep S of the operand `transaction/@value` are determined by the rules in [19.8.8 Streamability of Path Expressions](#), as follows:
 - A. The expression is expanded to `child::transaction/attribute::value`.
 - B. The posture and sweep of the left-hand operand `child::transaction` are determined by the rules in [19.8.8.9 Streamability of Axis Steps](#), as follows:
 - I. The context posture is striding, because the focus-setting container is the template rule itself.
 - II. The context item type is `element()`, based on the match type of the pattern `match="transactions[@currency='USD']"`.
 - III. Rules 1 and 2 do not apply because the context posture is striding.
 - IV. Rule 3 does not apply because the child axis applied to an element node is not necessarily empty.
 - V. Rule 4 does not apply because there are no predicates.
 - VI. Rule 5 applies, and the table entry with context posture = striding, axis = child gives a result posture of striding and a sweep of consuming.
 - C. The posture of the relative path expression `child::transaction/attribute::value` is therefore the posture of its right-hand operand `attribute::value`, assessed with a context posture of striding. This is determined by the rules in [19.8.8.9 Streamability of Axis Steps](#), as follows:
 - I. The context posture, as we have seen, is striding.
 - II. The context item type is `element()`, based on the type of the left-hand operand `child::transaction`.
 - III. Rules 1 and 2 do not apply because the context posture is striding.
 - IV. Rule 3 does not apply because the attribute axis applied to an element node is not necessarily empty.
 - V. Rule 4 does not apply because there are no predicates.
 - VI. Rule 5 applies, and the table entry with context posture = striding, axis = attribute gives a result posture of striding and a sweep of motionless.

- D. The posture of the relative path expression `child::transaction/attribute::value` is therefore striding.
- E. The sweep of the relative path expression `child::transaction/attribute::value` is the wider of the sweeps of its two operands, namely consuming and motionless. That is, it is consuming.
- v. So the first and only operand to the call on `sum()` has $U = \text{absorption}$, $T = \text{attribute}()$, $P = \text{climbing}$, and $S = \text{consuming}$
- vi. Rule 1(b) of the general streamability rules computes the adjusted sweep S' . Rule 1(b)(iii)(A) applies, so the effective operand usage U' is inspection. Rule 1(b)(iii)(A) then computes the adjusted sweep from the table entry for $P = \text{climbing}$, $U' = \text{inspection}$; this shows $S' = S$, that is, consuming.
- vii. Rule 2(d) now applies, so the call on `sum()` is grounded and consuming.
- e. Since the `xsl:value-of` instruction has one operand with $U = \text{absorption}$, $T = \text{xs:anyAtomicType}$, $P = \text{grounded}$, and $S = \text{consuming}$, rule 2(d) again applies, and the `xsl:value-of` instruction is grounded and consuming.
- 5. Since the literal result element has one operand with $U = \text{absorption}$, $T = \text{item}()$, $P = \text{grounded}$, and $S = \text{consuming}$, rule 2(d) again applies, and the literal result element is grounded and consuming.
- 6. Therefore the body of the template rule is grounded, and since the other conditions are also satisfied, it is guaranteed-streamable.

Example: Streamed Grouping

Consider the following code, which is designed to process a transaction file containing transactions in chronological order, and output the total value of the transactions for each day.

```
<xsl:template name="go">
<out>
  <xsl:source-document streamable="yes" href="transactions.xml">
    <xsl:for-each-group select="/account/transaction"
      group-adjacent="xs:date(@timestamp)">
      <total date="{current-grouping-key()}" value="{sum(current-
group()/@value)}"/>
    </xsl:for-each-group>
  </xsl:source-document>
</out>
</xsl:template>
```

The rules for [xsl:source-document](#) say that the instruction is [guaranteed-streamable](#) if the contained [sequence constructor](#) is [grounded](#), and the task of streamability analysis is to prove that this is the case. As in the previous example, we will take a short-cut by making the assumption that a sequence constructor containing a single instruction can be replaced by that instruction in the construct tree.

So the task is to show that the [xsl:for-each-group](#) instruction is [grounded](#), which we can do as follows:

1. The relevant rules are to be found in [19.8.4.19 Streamability of xsl:for-each-group](#).

Note:

Rule numbers may be different in a version of the specification with change markings.

2. Rule 1 applies only if the [select](#) expression is [grounded](#). It is easy to see informally that this is not the case (an expression that returns streamed nodes is never grounded). More formally:

- a. The [select](#) expression is a path expression; the rules in [19.8.8.8 Streamability of Path Expressions](#) apply.
- b. The expression is rewritten as `((root(.) treat as document-node())/child::account)/child::transaction`
- c. The left-hand operand `(root(.) treat as document-node())/child::account` is also a path expression, so the rules in [19.8.8.8 Streamability of Path Expressions](#) apply recursively:
 - i. The left-hand operand `root(.) treat as document-node()` follows the rules for a [TreatExpr](#) in [19.8.8 Classifying Expressions](#); the proforma `T treat as TYPE` indicates that the [general streamability rules](#) apply with a single operand having usage [transmission](#).
 - ii. This single operand `root(.)` follows the rules in [19.8.9.18 Streamability of the root Function](#). The item type of the operand `.` is the [context item type](#), which is the type established by the [xsl:source-document](#) instruction, namely `document-node()`. Under these conditions `root(.)` is rewritten as `.`, so the [posture](#) is the [context posture](#) established by the [xsl:source-document](#) instruction, namely [striding](#). The [sweep](#) is [motionless](#).
 - iii. The [posture](#) and [sweep](#) of the expression `root(.) treat as document-node()` are the same as the [posture](#) and [sweep](#) of `root(.)`, namely [striding](#) and [motionless](#)

- iv. The right-hand operand `child::account` is governed by the rules in [19.8.8.9 Streamability of Axis Steps](#). The `context posture` is `striding`, and the axis is `child`, so the result posture is `striding` and the sweep is `consuming`.
 - v. The `posture` of the path expression is the `posture` of the right-hand operand, that is `striding`, and its sweep is the wider sweep of the two operands, that is `consuming`
 - d. Returning to the outer path expression, the `posture` of the right hand operand `child::transaction` is `striding`, and its sweep is `consuming`.
 - e. So the `posture` of the `select` expression as a whole is the posture of the right hand operand, that is `striding`; and its sweep is the wider of the sweeps of the operands, which is `consuming`.
3. Rule 2 does not apply: there is no `group-by` attribute.
4. Rule 3 does not apply: there is a `group-adjacent` attribute, but it is `motionless`. The reasoning is as follows:
- a. The value is a call to the constructor function `xs:date`. The rules in [19.8.8.14 Streamability of Static Function Calls](#) apply. There is a single operand, whose required type is atomic, so the `operand usage` is `absorption`.
 - b. These rules refer to the [general streamability rules](#), so we need to determine the `context item type`, `posture`, and `sweep` of the operand expression `@timestamp`. This is done as follows:
 - i. The expression is an `AxisStep`, so the relevant rules are in [19.8.8.9 Streamability of Axis Steps](#).
 - ii. The `context posture` is the `posture` of the `controlling operand` of the `focus-setting container`, that is, is the `select` expression of the containing `xsl:for-each-group` instruction, which as established above is `striding`. The `context item type` is similarly the inferred type of the `select` expression, and is `element()`.
 - iii. Rules 1 and 2 do not apply because the `context posture` is `striding`.
 - iv. Rule 3 does not apply because the attribute axis for an element node is not necessarily empty.
 - v. Rule 4 does not apply because there is no predicate.
 - vi. So the `sweep` and `posture` of the expression `@timestamp` are given by the table in Rule 5 as `striding` and `motionless`.
 - c. Returning to the [general streamability rules](#) for the expression `xs:date(@timestamp)`, the operand `@timestamp` has $U = \text{absorption}$, $T = \text{attribute}()$, $P = \text{striding}$, $S = \text{motionless}$.
 - d. Under Rule 1(b)(iii)(A), because $T = \text{attribute}()$, the `operand usage` U' becomes `inspection`.
 - e. Under Rule 1(b)(iii)(A), $S' = S = \text{motionless}$.
 - f. Under Rule 2(e), the expression `xs:date(@timestamp)` is `grounded` and `motionless`.
5. Rule 4 (under `xsl:for-each-group`) does not apply, because there is no `xsl:sort` child.
6. So Rule 5 applies. This relies on knowing the `posture` of the sequence constructor contained in the `xsl:for-each-group` instruction: that is, the `posture` of the total `literal result element`. This is calculated as follows:
- a. The rules that apply are in [19.8.4.1 Streamability of Literal Result Elements](#). The [general streamability rules](#) apply; there are two operands, the attribute value templates `{current-grouping-key()}` and `{sum(current-group()/@value)}`, and in each case the usage is `absorption`. We can simplify the analysis by observing that the empty `sequence constructor` contained in the literal result element can be ignored, since it is `grounded` and `motionless`.

- b. Consider first the operand `{current-grouping-key()}`.
 - i. Section [19.8.7 Classifying Value Templates](#) applies. This refers to the [general streamability rules](#); there is a single operand, the expression `current-grouping-key()`, with usage [absorption](#).
 - ii. Section [19.8.9.5 Streamability of the current-grouping-key Function](#) applies. This establishes that the expression is [grounded](#) and [motionless](#).
 - iii. It follows that the operand `{current-grouping-key()}` expression is also [grounded](#) and [motionless](#).
- c. Now consider the operand `{sum(current-group()/@value)}`.
- d. Section [19.8.7 Classifying Value Templates](#) applies. This refers to the [general streamability rules](#); there is a single operand, the expression `sum(current-group()/@value)`, with usage [absorption](#).
- e. The rules for the `sum` function appear in [19.8.9 Classifying Calls to Built-In Functions](#). The proforma is given there as `fn:sum(A)`, which means that the [general streamability rules](#) apply, and that the single operand `current-group()/@value` has usage [absorption](#). So we need to establish the [posture](#), [sweep](#), and type of this expression, which we can do as follows:
 - i. The expression is a `RelativePathExpr`, so section [19.8.8.8 Streamability of Path Expressions](#) applies.
 - ii. The expression is expanded to `current-group()/attribute::value`.
 - iii. The [posture](#) and [sweep](#) of the left-hand operand `current-group()` are defined in [19.8.9.4 Streamability of the current-group Function](#). Since all the required conditions are satisfied, the [posture](#) of `current-group()` is the [posture](#) of the `select` expression, that is [striding](#), and its [sweep](#) is the [sweep](#) of the `select` expression, that is [consuming](#).
 - iv. The [posture](#) and [sweep](#) of the right hand operand `@value` are defined in [19.8.8.9 Streamability of Axis Steps](#). The [context posture](#) is the [posture](#) of the left-hand operand `current-group()`, namely [striding](#); the table in Rule 5 applies, giving the result [climbing](#) and [motionless](#).
 - v. The [posture](#) of the `RelativePathExpr` is the [posture](#) of the right hand operand, namely [striding](#). The [sweep](#) of the `RelativePathExpr` is the wider of the [sweeps](#) of its operands, which is [consuming](#).
 - vi. The type of the expression `current-group()/@value` is determined using the rules in [19.1 Determining the Static Type of a Construct](#) as `attribute()`.
- f. So the `sum` function has a single operand with $U = \text{absorption}$, $P = \text{striding}$, $S = \text{consuming}$, $T = \text{attribute}()$.
- g. In the [general streamability rules](#), Rule 1(b)(iii)(A) gives the adjusted usage as $U' = \text{inspection}$, and Rule 1(b)(iii)(B) gives the adjusted sweep as $S' = S = \text{consuming}$. Rule 2(d) gives the posture and sweep of the call to `sum` as [grounded](#) and [consuming](#).
- 7. So the literal result element has two operands, one of which is [grounded](#) and [motionless](#), the other [grounded](#) and [consuming](#). Rule 2(d) of the [general streamability rules](#) determines that the literal result element is [grounded](#) and [consuming](#).
- 8. So the content of the `xsl:source-document` instruction is [grounded](#), which means that the instruction is [guaranteed-streamable](#).

19.10 Streamability Guarantees

Certain constructs allow a stylesheet author to declare that a construct is streamable. Specifically:

- Specifying `streamable="yes"` on `xsl:mode` declares that all template rules in that mode (and all template rules that specify `mode="#all"`) are streamable;
- Specifying `streamable="yes"` on `xsl:source-document` declares that its contained sequence constructor is streamable;
- Specifying `streamable="yes"` on `xsl:function` declares that the `stylesheet function` in question is streamable;
- Specifying `streamable="yes"` on `xsl:attribute-set` declares that the attribute set in question is streamable;
- Specifying `streamable="yes"` (explicitly or implicitly) on `xsl:merge-source` declares that the merging process is streamable with respect to that particular input.
- Specifying `streamable="yes"` on `xsl:accumulator` declares that the accumulator can be evaluated on a streamed document.

[**DEFINITION:** The above constructs (template rules belonging to a mode declared with `streamable="yes"`; and `xsl:source-document`, `xsl:attribute-set`, `xsl:function`, `xsl:merge-source`, and `xsl:accumulator` elements specifying `streamable="yes"`) are said to be **declared-streamable**.]

In each case the construct in question is said to be **guaranteed-streamable** if it satisfies two conditions:

1. The construct is **declared-streamable**.
2. Streamability analysis following the rules defined in this specification determines that streamed processing is possible (the detailed conditions vary from one construct to another).

[**DEFINITION:** A **guaranteed-streamable** construct is a `construct` that is declared to be streamable and that follows the particular rules for that construct to make streaming possible, as defined by the analysis in this specification.]

For a streaming processor, that is, a processor that claims conformance with the `streaming feature`:

1. If a construct is **guaranteed-streamable** and the input is provided in streamable form, then the input **MUST** be processed using streaming.

Note:

The requirement to process the input using streaming does not apply if the processor is able to determine that this would convey no benefit: for example, if the input is supplied as a tree in memory. However, this does not remove the requirement to verify that the relevant stylesheet constructs are **guaranteed-streamable**.

2. If a construct is declared as streamable but is not **guaranteed-streamable** (that is, if it fails to satisfy the conditions for streamability defined in this specification), then the processor **MUST** be prepared to do any one of the following at user option:
 - a. Signal a static error [see [ERR_XTSE3430](#)]
 - b. Process the stylesheet as if it were a non-streaming processor (see below)

- c. Process the stylesheet with streaming if it is able to do so, or signal a static error [see [ERR_XTSE3430](#)] if it is not able to do so.

[ERR_XTSE3430] It is a [static error](#) if a [package](#) contains a construct that is declared to be streamable but which is not [guaranteed-streamable](#), unless the user has indicated that the processor is to handle this situation by processing the stylesheet without streaming or by making use of processor extensions to the streamability rules where available.

For a non-streaming processor, the processor **MUST** evaluate the construct delivering the same results as if execution used streaming, but with no constraints on the evaluation strategy. (Processing **MAY**, of course, fail due to insufficient memory being available, or for other reasons.) A non-streaming processor is **NOT REQUIRED** to assess whether constructs are [guaranteed-streamable](#), or to apply restrictions such as the rules for where calls on the functions [accumulator-before](#) and [accumulator-after](#) may appear. However, a non-streaming processor **MUST** enforce the constraint implied by a [use-accumulators](#) attribute restricting which accumulators can be used with a particular document.

Note:

This specification does not attempt to legislate precisely what constitutes evaluation “using streaming”. The most important test is that the amount of memory needed should be for practical purposes independent of the size of the source document, and in particular that the finite size of memory available should not impose a limit on the size of source document that can be processed.

The rules are designed to ensure that streaming processors can analyze streamability using rules different from those in this specification, provided that all constructs that are [guaranteed-streamable](#) according to this specification are actually streamable by the implementation. Furthermore, non-streaming processors are not required to analyze streamability at all.

20 Additional Functions

This section describes XSLT-specific additions to the XPath function library. Some of these additional functions also make use of information specified by [declarations](#) in the stylesheet; this section also describes these declarations.

20.1 [fn:document](#)

Summary

Provides access to XML documents identified by a URI.

Signatures

```
fn:document($uri-sequence as item()* as node())*
```

```
fn:document($uri-sequence as item(),
            $base-node      as node()) as node()*
```

Properties

The one-argument form of this function is [deterministic](#)^{FO30}, [focus-independent](#)^{FO30}, and [context-dependent](#)^{FO30}. It depends on static base URI.

The two-argument form of this function is [deterministic](#)^{FO30}, [focus-independent](#)^{FO30}, and [context-independent](#)^{FO30}.

Rules

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 `$uri-sequence` that is an instance of `xs:string`, `xs:anyURI`, or `xs:untypedAtomic`, the value is cast to `xs:anyURI`. If the resulting URI reference is an absolute URI reference then it is used *as is*. If it is a relative URI reference, then it is resolved as follows:
 1. If `$base-node` is supplied, then it is resolved against the base URI of `$base-node`.
 2. Otherwise it is resolved against the static base URI from the static context of the expression containing the call to the [document](#) function. In cases where the source code of the stylesheet is available at execution time, this will typically be the location of the relevant stylesheet module.
- For an item in `$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 `$base-node` if supplied, or against the base URI of the node that contained it otherwise.
- A relative URI is resolved against a base URI using the rules of the [resolve-uri](#)^{FO30} function. A dynamic error occurs (see below) if no base URI is available.
- If `$uri-sequence` (after atomizing any nodes) contains an item other than an atomic value of type `xs:string`, `xs:anyURI`, or `xs:untypedAtomic` then a type error is raised [\[ERR_XPTY0004\]](#)^{XP30}.

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](#)^{FO30} function defined in [\[Functions and Operators 3.0\]](#). This returns a document node. If an error occurs during evaluation of the [doc](#)^{FO30} 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.

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.

Error Conditions

[ERR XTDE1160] When a URI reference contains a fragment identifier, it is a [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).

A processor MAY provide an option which, if selected, causes the processor instead of signaling this error, to ignore the fragment identifier and return the document node.

The set of media types recognized by a processor is [implementation-defined](#).

[ERR XTDE1162] When a URI reference is a relative reference, it is a [dynamic error](#) if no base URI is available to resolve the relative reference. This can arise for example when the URI is contained in a node that has no base URI (for example a parentless text node), or when the second argument to the function is a node that has no base URI, or when the base URI from the static context is undefined.

Notes

One effect of these rules is that in an interpreted environment where the source code of the stylesheet is available and its base URI is known, then unless XML entities or `xml:base` are used, the expression `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 then 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.

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.

20.2 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.

20.2.1 [The `xsl:key` Declaration](#)

```
<!-- Category: declaration -->
<xsl:key
  name = eqname
  match = pattern
  use? = expression
  composite? = boolean
  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 an [EQName](#), which is expanded as described in [5.1.1 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 same [package](#) that share the same name.]

The key name is scoped to the containing [package](#), and is available for use in calls to the [key](#) function within that package.

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 a [singleton focus](#) based on the node that matches the pattern. 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](#).]

When evaluation of the [key specifier](#) results in a sequence (after atomization) containing more than one atomic value, the effect depends on the value of the composite attribute:

- When the attribute is absent or has the value no, each atomic value in the sequence acts as an individual key. For example, if `match="book" use="author" composite="no"` is specified, then a book element may be located using the value of any author element.
- When the attribute is present and has the value yes, the sequence of atomic values is treated as a composite key that must be matched in its entirety. For example, if `match="book" use="author" composite="yes"` is specified, then a book element may be located using the value of all its author elements, supplied in the correct order.

If there are several [xsl:key](#) declarations in the same package with the same key name, then they must all have the same effective value for their composite attribute. The effective value is the actual value of the attribute if present, or "no" if the attribute is absent.

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 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:

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.

Keys can only be used to search within a tree that is rooted at a document node.

The optional `collation` attribute is used only when deciding whether two strings are equal for the purposes of key matching. Specifically, two key values `$a` and `$b` are considered equal if the result of the function call `deep-equal($a, $b, $collation)` is true. 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 same `package` 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 XTSE1222] It is a `static error` if there are several `xsl:key` declarations in a `package` with the same key name and different effective values for the `composite` attribute.

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` (which can be treated either as separate individual key values, or as a single composite key value);
- 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.

20.2.2 `fn:key`

Summary

Returns the nodes that match a supplied key value.

Signatures

```
fn:key($key-name as xs:string,
       $key-value as xs:anyAtomicType*) as node()*
```

```
fn:key($key-name as xs:string,
       $key-value as xs:anyAtomicType*,
       $top       as node()*)
```

Properties

The two-argument form of this function is [deterministic^{FO30}](#), [focus-dependent^{FO30}](#), and [context-dependent^{FO30}](#).

The three-argument form of this function is [deterministic^{FO30}](#), [focus-independent^{FO30}](#), and [context-dependent^{FO30}](#).

Rules

The [key](#) function does for keys what the [element-with-id^{FO30}](#) function does for IDs.

The \$key-name argument specifies the name of the [key](#). The value of the argument MUST be a string containing an [EQName](#). If it is a [lexical QName](#), then it is expanded as described in [5.1.1 Qualified Names](#) (no prefix means no namespace).

The \$key-value argument to the [key](#) function is considered as a sequence. The effect depends on the value of the composite attribute of the corresponding [xsl:key](#) declaration.

- If composite is no or absent, 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.

If the second argument is an empty sequence, the result of the function will be an empty sequence.

- If composite is yes, the requested key value is the sequence formed by atomizing the supplied value of the argument, using the standard [function conversion rules](#). 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](#) is deep-equal to the requested key value, under the rules appropriate to the [deep-equal^{FO30}](#) function applied to the two values in question, using the [collation](#) attributes of the [xsl:key](#) declaration when comparing strings. Note that the [deep-equal^{FO30}](#) function reports no error if two values are encountered that are not comparable; they are regarded for the purposes of this function as being not equal.

If the second argument is an empty sequence, the result of the function will be the set of nodes having an empty sequence as the value of the key specifier.

Different rules apply when [XSLT 1.0 compatible behavior](#) is enabled.

A key (that is, a set of [xsl:key](#) declarations sharing the same key name) is processed in backwards compatible mode if (a) at least one of the [xsl:key](#) elements in the definition of the key enables backwards compatible behavior, and (b) the effective value of the composite attribute is no.

When a key is processed in backwards compatible mode, then:

- The result of evaluating the key specifier in any `xsl:key` declaration having this key name is converted after atomization to a sequence of strings, by applying a cast to each item in the sequence.
- When the first argument to the `key` function specifies this key name, then the value of the second argument is converted after atomization to a sequence of strings, by applying a cast to each item in the sequence. The values are then compared as strings.

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 `/`.

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 `composite="no"`, and 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.

When `composite="yes"`, and the `key specifier` of that `xsl:key` declaration is evaluated with a `singleton focus` based on `$N`, the `atomized` value of the resulting sequence compares equal to the atomized value of the sequence supplied as `$key-value`, under the rules of the `deep-equal`^{FO30} function 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.

Error Conditions

[ERR XTDE1260] It is a `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 containing `package`. 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 `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.

Notes

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.

Examples

Example: Using a Key to Follow Cross-References

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.

Example: Using a Key to Generate Hyperlinks

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>
```

Example: Using a Composite Key

This example uses a composite key consisting of first name and last name to locate employees in an employee file.

The key can be defined like this:

```
<xsl:key name="emp-name-key"
         match="employee"
         use="name/first, name/last"
         composite="yes"/>
```

A particular employee can then be located using the function call:

```
key('emp-name-key', ('Tim', 'Berners-Lee'), doc('employees.xml'))
```

20.3 Keys and Streaming

Keys are not applicable to streamed documents.

This is ensured by the rules for the streamability of the [key](#) function (see [19.8.9 Classifying Calls to Built-In Functions](#)). These rules make the [operand usage](#) of the third argument [navigation](#), which has the consequence that when the [key](#) function is applied to a streamed input document, the call is [roaming](#) and [free-ranging](#), which effectively makes the containing construct non-streamable.

20.4 Miscellaneous Additional Functions

20.4.1 [fn:current](#)

Summary

Returns the item that is the context item for the evaluation of the containing XPath expression

Signature

```
fn:current() as item()
```

Properties

This function is [deterministic^{FO30}](#), [context-dependent^{FO30}](#), and [focus-dependent^{FO30}](#).

Rules

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.

If the [current](#) function is used within a [pattern](#), its value is the item that is being matched against the pattern.

Error Conditions

[ERR XTDE1360] If the [current](#) function is evaluated within an expression that is evaluated when the context item is absent, a [dynamic error](#) occurs.

When the [current](#) is called by means of a dynamic function call (for example, `current#0()`), it is evaluated as if the context item is absent ([see [ERR XTDE1360](#)]).

Examples

The instruction:

```
<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.

20.4.2 `fn:unparsed-entity-uri`

Summary

Returns the URI (system identifier) of an unparsed entity

Signatures

```
fn:unparsed-entity-uri($entity-name as xs:string) as xs:anyURI
```

```
fn:unparsed-entity-uri($entity-name as xs:string,
                      $doc           as node()) as xs:anyURI
```

Properties

This function is `deterministic`^{FO30}, `focus-dependent`^{FO30}, and `context-dependent`^{FO30}.

Rules

Calling the single-argument form of this function has the same effect as calling the two-argument form with the context item as the second argument.

The two-argument `unparsed-entity-uri` function returns the URI of the unparsed entity whose name is given by the value of the `$entity-name` argument, in the document containing the node supplied as the value of the `$doc` argument. 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 [\[XDM 3.0\]](#).

Error Conditions

[ERR XTDE1370] It is a `dynamic error` if `$node`, or the context item if the second argument is omitted, is a node in a tree whose root is not a document node.

The following errors may be raised when `$node` is omitted:

- If the context item is absent, `dynamic error [ERR XPDY0002]`^{XP30}.
- If the context item is not a node, `type error [ERR XPTY0004]`^{XP30}.

Notes

The XDM accessor `dm:unparsed-entity-system-id` is defined to return an absolute URI, obtained by resolving the system identifier as written against the base URI of the document. If no base URI is available for the document, the `unparsed-entity-uri` function SHOULD return the system identifier as written, without any attempt to make it absolute.

XML permits more than one unparsed entity declaration with the same name to appear, and says that the first declaration is the one that should be used. This rule SHOULD be respected during construction of the data model; the data model instance should not contain more than one unparsed entity with the same name.

20.4.3 `fn:unparsed-entity-public-id`

Summary

Returns the public identifier of an unparsed entity

Signatures

```
fn:unparsed-entity-public-id($entity-name as xs:string) as xs:string
```

```
fn:unparsed-entity-public-id($entity-name as xs:string,
                             $doc           as node()) as xs:string
```

Properties

This function is `deterministic`^{FO30}, `focus-dependent`^{FO30}, and `context-dependent`^{FO30}.

Rules

Calling the single-argument form of this function has the same effect as calling the two-argument form with the context item as the second argument.

The two-argument `unparsed-entity-public-id` function returns the public identifier of the unparsed entity whose name is given by the value of the `$entity-name` argument, in the document containing the node supplied as the value of the `$doc` argument. 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 [\[XDM 3.0\]](#).

Error Conditions

[ERR XTDE1380] It is a `dynamic error` if `$node`, or the context item if the second argument is omitted, is a node in a tree whose root is not a document node.

The following errors may be raised when `$node` is omitted:

- If the context item is absent, `dynamic error [ERR XPDY0002]`^{XP30}.
- If the context item is not a node, `type error [ERR XPTY0004]`^{XP30}.

Notes

XML permits more than one unparsed entity declaration with the same name to appear, and says that the first declaration is the one that should be used. This rule SHOULD be respected during construction of the data model; the data model instance should not contain more than one unparsed entity with the same name.

20.4.4 [fn:system-property](#)

Summary

Returns the value of a system property

Signature

```
fn:system-property($property-name as xs:string) as xs:string
```

Properties

This function is [deterministic](#)^{FO30}, [focus-independent](#)^{FO30}, and [context-dependent](#)^{FO30}. It depends on namespaces.

Rules

The value of the \$property-name argument MUST be a string containing an [E QName](#). If it is a [lexical QName](#) with a prefix, then it is expanded into an [expanded QName](#) using the namespace declarations in the static context of the [expression](#). If there is no prefix, the name is taken as being in no namespace.

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 "3.0". The value will always be a string in the lexical space of the decimal datatype 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 [XSLT 1.0 compatibility feature](#), or "no" otherwise.
- `xsl:supports-namespace-axis`, returns the string "yes" in the case of a processor that offers the XPath namespace axis even when not in backwards compatible mode, or "no" otherwise. Note that a processor that supports backwards compatible mode must support the namespace axis when in that mode, so this property is not relevant to that case.

- `xsl:supports-streaming`, returns the string "yes" in the case of a processor that offers the streaming feature (see [27.5 Streaming Feature](#)), or "no" otherwise.
- `xsl:supports-dynamic-evaluation`, returns the string "yes" in the case of a processor that offers the dynamic evaluation feature (see [27.6 Dynamic Evaluation Feature](#)), or "no" otherwise.
- `xsl:supports-higher-order-functions`, returns the string "yes" in the case of a processor that offers the [higher-order functions feature](#), or "no" otherwise.
- `xsl:xpath-version`, a number giving the version of XPath implemented by the [processor](#). The value will always be a string in the lexical space of the decimal datatype 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. Typical values are "3.0" or "3.1". The value "3.0" indicates that the processor implements XPath 3.0 plus the extensions defined in [21 Maps](#) and [22 Processing JSON Data](#).
- `xsl:xsd-version`, a number giving the version of XSD (XML Schema) implemented by the [processor](#). The value will always be a string in the lexical space of the decimal datatype 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. Typical values are "1.0" or "1.1". This property is relevant even when the processor is not schema-aware, since the built-in datatypes for XSD 1.1 differ from those in XSD 1.0.

Some of these properties relate to the conformance levels and features offered by the [processor](#): these options are described in [27 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.

Error Conditions

[ERR XTDE1390] It is a [dynamic error](#) if the value supplied as the `$property-name` argument 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](#).

Notes

An implementation must not return the value 3.0 as the value of the `xsl:version` system property unless it is conformant to XSLT 3.0.

It is recognized that vendors who are enhancing XSLT 1.0 or 2.0 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 3.0, 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 3.0, and to provide the [element-available](#) and [function-available](#) functions to allow users to test which features have been fully implemented.

20.4.5 [fn:available-system-properties](#)

Summary

Returns a list of system property names that are suitable for passing to the [system-property](#) function, as a sequence of QNames.

Signature

```
fn:available-system-properties() as xs:QName*
```

Properties

This function is deterministic^{FO30}, context-independent^{FO30}, and focus-independent^{FO30}.

Rules

The function returns a sequence of QNames, being the names of the system properties recognized by the processor, in some implementation-dependent order.

The prefix part of a returned QName is implementation-dependent.

The function is deterministic^{FO30}: that is, the set of available system properties does not vary during the course of a transformation.

Notes

The function returns a list of QNames, containing no duplicates.

It is intended that the QNames in this list should be suitable for passing to system-property. However, they must first be converted to the form expected by the system-property function, which is either a lexical QName or to an EQName in the form `Q{uri}local`. Because the prefix of the returned QName is unpredictable, the `Q{uri}local` is likely to be more convenient. Conversion of an `xs:QName` value to an EQName in `Q{uri}local` format can be achieved using the function:

```
<xsl:function name="f:QName-to-brace-notation" as="xs:string">
  <xsl:param name="qname" as="xs:QName"/>
  <xsl:sequence select="'Q{' || namespace-uri-from-QName($qname) || '}'
    || local-name-from-QName($qname)"/>
</xsl:function>
```

21 Maps

When XSLT 3.0 is used with XPath 3.0, it extends the type system and data model of XPath 3.0 with an additional datatype: the map. A map is an additional kind of item. Supporting this additional type are additional XPath language constructs, types, and XSLT instructions, all defined in this section.

Note:

The extensions to XPath 3.0 defined in this section have been incorporated into XPath 3.1. Therefore, when an XSLT 3.0 processor implements the XPath 3.1 Feature, the relevant parts of this section can be ignored.

[**DEFINITION:** A map consists of a set of entries. Each entry comprises a key which is an arbitrary atomic value, and an arbitrary sequence called the associated value.]

[**DEFINITION:** Within a map, no two entries have the **same key**. Two atomic values K1 and K2 are the **same key** for this purpose if the relation `op:same-key(K1, K2, $UCC)` holds.]

To put it another way, subject to the rule above regarding timezones, the keys are the same if either `K1 eq K2` is true under the Unicode codepoint collation, or if both `K1` and `K2` are `NaN`. It is not necessary that all the keys should be mutually comparable (for example, they can include a mixture of integers and strings).

The function call `map:get($map, $key)` can be used to retrieve the value associated with a given key.

A `map` can also be viewed as a function from keys to associated values. To achieve this, a map is also a function item. The properties of this function are as follows:

- The **name** of the function is absent.
- The **arity** of the function is 1 (one).
- The **parameter names** comprise a sequence of one QName, conventionally `$key`, though the choice of name has no observable consequences.
- The **signature** is `function($key as xs:anyAtomicValue) as item()*` (with no annotations).
- The **implementation** is the expression `map:get($self, $key)`
- The **non-local-variable-bindings** comprise a single variable, `$self`, whose value is the map itself.

Calling the function has the same effect as calling the `get` function: the expression `$map($key)` returns the same result as `get($map, $key)`. For example, if `$books-by-isbn` is a map whose keys are ISBNs and whose associated values are book elements, then the expression `$books-by-isbn("0470192747")` returns the book element with the given ISBN. The fact that a map is a function item allows it to be passed as an argument to higher-order functions that expect a function item as one of their arguments.

Like all other values, `maps` are immutable. For example, the `map:remove` function returns a map that differs from the supplied map by the omission of one entry, but the supplied map is not changed by the operation. Two calls on `map:remove` with the same arguments will return maps that are indistinguishable from each other; there is no way of asking whether these are “the same map”.

21.1 The Type of a Map

The syntax of `ItemTypeXP30` as defined in XPath is extended as follows:

MapType

```
[69] ItemType ::= KindTest | ("item" "(" ")") | FunctionTest | AtomicOrUnionType |
      ParenthesizedItemType
      | MapType
[201] MapType ::= 'map' '(' '*' | (AtomicOrUnionTypeXP30 ',' SequenceTypeXP30) ')'
```

The following rules express the matching rules for a map item type and a map, and extend the set of rules given in [Section 2.5.5.2 Matching an ItemType and an Item^{XP30}](#):

- The `ItemType map(K, V)` matches an item `M` if (a) `M` is a `map`, and (b) every entry in `M` has a key that matches `K` and an associated value that matches `V`. For example, `map(xs:integer, element(employee))` matches a map if all the keys in the map are integers, and all the associated values are `employee` elements. Note that a map (like a sequence) carries no intrinsic type information separate from the types of its entries,

and the type of existing entries in a map does not constrain the type of new entries that can be added to the map.

Note:

In consequence, `map(K, V)` matches an empty map, whatever the types *K* and *V* might be.

- The ItemType `map(*)` matches any map regardless of its contents. It is equivalent to `map(xs:anyAtomicType, item())*`.

A map also acts as a function. This means that maps match certain function item types. Specifically, the following rule extends the list of rules in [Section 2.5.5.7 Function Test](#)^{XP30}:

- `function(*)` matches any map.
- `function(xs:anyAtomicType) as item()*` matches any map.

Because of the rules for subtyping of function types according to their signature, it follows that the item type `function(A) as item()*`, where *A* is an atomic type, also matches any map, regardless of the type of the keys actually found in the map. For example, a map whose keys are all strings can be supplied where the required type is `function(xs:integer) as item()*`; a call on the map that treats it as a function with an integer argument will always succeed, and will always return an empty sequence.

The function signature of the map, treated as a function, is always `function(xs:anyAtomicType) as item()*`, regardless of the actual types of the keys and values in the map. This means that a function item type with a more specific return type, such as `function(xs:anyAtomicType) as xs:integer`, does not match a map in the sense required to satisfy the `instance` of operator. However, the rules for function coercion mean that any map can be supplied as a value in a context where such a type is the required type, and a type error will only occur if an actual call on the map (treated as a function) returns a value that is not an instance of the required return type.

Note:

So, given a map \$M whose keys are integers and whose results are strings, such as `map{0:"no", 1:"yes"}`, the following relations hold, among others:

- \$M instance of `map(*)`
- \$M instance of `map(xs:integer, xs:string)`
- \$M instance of `map(xs:decimal, xs:anyAtomicType)`
- `not($M instance of map(xs:int, xs:string))`
- `not($M instance of map(xs:integer, xs:token))`
- \$M instance of `function(*)`
- \$M instance of `function(xs:anyAtomicType) as item()*`
- \$M instance of `function(xs:integer) as item()*`
- \$M instance of `function(xs:int) as item()*`
- \$M instance of `function(xs:string) as item()*`
- `not($M instance of function(xs:integer) as xs:string)`

The last case might seem surprising; however, function coercion ensures that \$M can be used successfully anywhere that the required type is `function(xs:integer) as xs:string`.

The rules for judging whether one item type is a subtype of another, given in [Section 2.5.6.2 The judgement subtype-itemtype\(A_i, B_i\)^{XP30}](#), are extended with some additional rules. The judgement `subtype-itemtype(Ai, Bi)` is true if:

- A_i is `map(K, V)` and B_i is `map(*)`, for any K and V.
- A_i is `map(Ka, Va)` and B_i is `map(Kb, Vb)`, where `subtype-itemtype(Ka, Kb)` and `subtype(Va, Vb)`.
- A_i is `map(*)` (or, because of the transitivity rules, any other map type) and B_i is `function(*)`.
- A_i is `map(*)`, (or, because of the transitivity rules, any other map type) and B_i is `function(xs:anyAtomicType) as item()*`.

[21.2 Functions that Operate on Maps](#)

XSLT 3.0 provides a number of functions that operate on maps, or that are useful in conjunction with maps. These functions are specified in [\[Functions and Operators 3.1\]](#), but they are available with XSLT 3.0 whether or not the processor offers the [XPath 3.1 Feature](#).

Some of the functions defined in this section use a conventional namespace prefix `map`, which is assumed to be bound to the namespace URI `http://www.w3.org/2005/xpath-functions/map`.

Note that there is no operation to atomize a map or convert it to a string.

[21.2.1 op:same-key](#)

Summary

Determines whether two atomic values can coexist as separate keys within a map.

Signature

```
op:same-key($k1 as xs:anyAtomicType,
            $k2 as xs:anyAtomicType) as xs:boolean
```

Properties

This function is deterministic^{FO30}, context-independent^{FO30}, and focus-independent^{FO30}.

Rules

The internal function `op:same-key` (which is not available at the user level) is used to assess whether two atomic values are considered to be duplicates when used as keys in a map. A map cannot contain two separate entries whose keys are **the same** as defined by this function. The function is also used when matching keys in functions such as `map:get` and `map:remove`.

The function returns true if and only if one of the following conditions is true:

1. All of the following conditions are true:

- a. $\$k1$ is an instance of `xs:string`, `xs:anyURI`, or `xs:untypedAtomic`
- b. $\$k2$ is an instance of `xs:string`, `xs:anyURI`, or `xs:untypedAtomic`
- c. `fn:codepoint-equal($k1, $k2)`

Note:

Strings are compared without any dependency on collations.

2. All of the following conditions are true:

- a. $\$k1$ is an instance of `xs:decimal`, `xs:double`, or `xs:float`
- b. $\$k2$ is an instance of `xs:decimal`, `xs:double`, or `xs:float`
- c. One of the following conditions is true:

- i. Both $\$k1$ and $\$k2$ are NaN

Note:

`xs:double('NaN')` is the same key as `xs:float('NaN')`

- ii. Both $\$k1$ and $\$k2$ are positive infinity

Note:

`xs:double('INF')` is the same key as `xs:float('INF')`

- iii. Both $\$k1$ and $\$k2$ are negative infinity

Note:

`xs:double('-INF')` is the same key as `xs:float('-INF')`

- iv. $\$k1$ and $\$k2$ when converted to decimal numbers with no rounding or loss of precision are mathematically equal.

Note:

Every instance of `xs:double`, `xs:float`, and `xs:decimal` can be represented exactly as a decimal number provided enough digits are available both before and after the decimal point. Unlike the `eq` relation, which converts both operands to `xs:double` values, possibly losing precision in the process, this comparison is transitive.

Note:

Positive and negative zero are the same key.

3. All of the following conditions are true:

- a. $\$k1$ is an instance of `xs:date`, `xs:time`, `xs:dateTime`, `xs:gYear`, `xs:gYearMonth`, `xs:gMonth`, `xs:gMonthDay`, or `xs:gDay`
- b. $\$k2$ is an instance of `xs:date`, `xs:time`, `xs:dateTime`, `xs:gYear`, `xs:gYearMonth`, `xs:gMonth`, `xs:gMonthDay`, or `xs:gDay`
- c. One of the following conditions is true:
 - i. Both $\$k1$ and $\$k2$ have a timezone
 - ii. Neither $\$k1$ nor $\$k2$ has a timezone
- d. `fn:deep-equal($k1, $k2)`

Note:

The use of `deep-equal` rather than `eq` ensures that comparing values of different types yields `false` rather than an error.

Note:

Unlike the `eq` operator, this comparison has no dependency on the implicit timezone, which means that the question of whether or not a map contains duplicate keys is not dependent on this aspect of the dynamic context.

4. All of the following conditions are true:

- a. $\$k1$ is an instance of `xs:boolean`, `xs:hexBinary`, `xs:base64Binary`, `xs:duration`, `xs:QName`, or `xs:NOTATION`
- b. $\$k2$ is an instance of `xs:boolean`, `xs:hexBinary`, `xs:base64Binary`, `xs:duration`, `xs:QName`, or `xs:NOTATION`
- c. `fn:deep-equal($k1, $k2)`

Note:

The use of `deep-equal` rather than `eq` ensures that comparing values of different types yields `false` rather than an error.

Notes

The rules for comparing keys in a map are chosen to ensure that the comparison is:

- **Context-free:** there is no dependency on the static or dynamic context
- **Error-free:** any two atomic values can be compared, and the result is either true or false, never an error
- **Transitive:** if A is the same key as B, and B is the same key as C, then A is the same key as C.

As always, any algorithm that delivers the right result is acceptable. For example, when testing whether an `xs:double` value D is the same key as an `xs:decimal` value that has N significant digits, it is not necessary to know all the digits in the decimal expansion of D to establish the result: computing the first $N+1$ significant digits (or indeed, simply knowing that there are more than N significant digits) is sufficient.

21.2.2 `map:merge`

Summary

Returns a map that combines the entries from a number of existing maps.

Signatures

```
map:merge($maps as map(*)* ) as map(*)
```

```
map:merge($maps      as map(*)*,
           $options as map(*)) as map(*)
```

Properties

This function is [deterministic^{FO30}](#), [context-independent^{FO30}](#), and [focus-independent^{FO30}](#).

Rules

The function `map:merge` returns a map that is formed by combining the contents of the maps supplied in the `$maps` argument.

Informally, the supplied maps are combined as follows:

1. There is one entry in the returned map for each distinct key present in the union of the input maps, where two keys are distinct if they are not the [same key](#).
2. If there are duplicate keys, that is, if two or more maps contain entries having the [same key](#), then the way this is handled is controlled by the second (`$options`) argument.

The definitive specification is as follows.

1. The effect of calling the single-argument function is the same as the effect of calling the two-argument function with an empty map as the value of `$options`.
2. The `$options` argument can be used to control the way in which duplicate keys are handled. The [option parameter conventions](#) apply.
3. The entries that may appear in the `$options` map are as follows:

Key	Value	Meaning
-----	-------	---------

duplicates	Determines the policy for handling duplicate keys: specifically, the action to be taken if two maps in the input sequence <code>\$maps</code> contain entries with key values K_1 and K_2 where K_1 and K_2 are the same key . The required type is <code>xs:string</code> . The default value is <code>use-first</code> .
reject	An error is raised [ERR FOJS0003] ^{F031} if duplicate keys are encountered.
use-first	If duplicate keys are present, all but the first of a set of duplicates are ignored, where the ordering is based on the order of maps in the <code>\$maps</code> argument.
use-last	If duplicate keys are present, all but the last of a set of duplicates are ignored, where the ordering is based on the order of maps in the <code>\$maps</code> argument.
combine	If duplicate keys are present, the result map includes an entry for the key whose associated value is the sequence-concatenation of all the values associated with the key, retaining order based on the order of maps in the <code>\$maps</code> argument. The key value in the result map that corresponds to such a set of duplicates must be the same key as each of the duplicates, but it is otherwise unconstrained: for example if the duplicate keys are <code>xs:byte(1)</code> and <code>xs:short(1)</code> , the key in the result could legitimately be <code>xs:long(1)</code> .
unspecified	If duplicate keys are present, all but one of a set of duplicates are ignored, and it is implementation-dependent which one is retained.

The result of the function call `map:merge($MAPS, $OPTIONS)` is defined to be consistent with the result of the expression:

```

let $FOJS0003 := QName("http://www.w3.org/2005/xqt-errors", "FOJS0003"),

$duplicates-handler := map {
  "use-first": function($a, $b) {$a},
  "use-last": function($a, $b) {$b},
  "combine": function($a, $b) {$a, $b},
  "reject": function($a, $b) {fn:error($FOJS0003)},
  "unspecified": function($a, $b) {fn:random-number-generator()?permute(($a, $b))
}},

$combine-maps := function($A as map(*), $B as map(*), $deduplicator as function(
  fn:fold-left(map:keys($B), $A, function($z, $k){
    if (map:contains($z, $k))
      then map:put($z, $k, $deduplicator($z($k), $B($k)))
    else map:put($z, $k, $B($k))
  })
)
return fn:fold-left($MAPS, map{}, 
  $combine-maps(?, ?, $duplicates-handler(($OPTIONS?duplicates, "use-first")[])
}

```

Note:

By way of explanation, `$combine-maps` is a function that combines two maps by iterating over the keys of the second map, adding each key and its corresponding value to the first map as it proceeds. The second call of `fn:fold-left` in the `return` clause then iterates over the maps supplied in the call to `map:merge`, accumulating a single map that absorbs successive maps in the input sequence by calling `$combine-maps`.

This algorithm processes the supplied maps in a defined order, but processes the keys within each map in implementation-dependent order.

The use of `fn:random-number-generator` represents one possible conformant implementation for "`duplicates`" : "`unspecified`", but it is not the only conformant implementation and is not necessarily a realistic implementation.

Error Conditions

An error is raised [\[ERR FOJS0003\]](#)^{FO31} if the value of `$options` indicates that duplicates are to be rejected, and a duplicate key is encountered.

An error is raised [\[ERR FOJS0005\]](#)^{FO31} if the value of `$options` includes an entry whose key is defined in this specification, and whose value is not a permitted value for that key.

Notes

If the input is an empty sequence, the result is an empty map.

If the input is a sequence of length one, the result map is indistinguishable from the supplied map.

There is no requirement that the supplied input maps should have the same or compatible types. The type of a map (for example `map(xs:integer, xs:string)`) is descriptive of the entries it currently contains, but is not a constraint on how the map may be combined with other maps.

Examples

```
let $week := map{0:"Sonntag", 1:"Montag", 2:"Dienstag",
  3:"Mittwoch", 4:"Donnerstag", 5:"Freitag",
  6:"Samstag"}
```

The expression `map:merge()` returns `map{}`. (*Returns an empty map*).

The expression `map:merge((map:entry(0, "no"), map:entry(1, "yes")))` returns `map{0:"no", 1:"yes"}`. (*Returns a map with two entries*).

The expression `map:merge(($week, map{7:"Unbekannt"}))` returns `map{0:"Sonntag", 1:"Montag", 2:"Dienstag", 3:"Mittwoch", 4:"Donnerstag", 5:"Freitag", 6:"Samstag", 7:"Unbekannt"}`. (*The value of the existing map is unchanged; the returned map contains all the entries from \$week, supplemented with an additional entry*.)

The expression `map:merge(($week, map{6:"Sonnabend"}), map{"duplicates":"use-last"})` returns `map{0:"Sonntag", 1:"Montag", 2:"Dienstag", 3:"Mittwoch", 4:"Donnerstag", 5:"Freitag", 6:"Sonnabend"}`. (*The value of the existing map is unchanged; the returned map contains all the entries from \$week, with one entry replaced by a new entry. Both input maps contain an entry with the key 6; the one used in the result is the one that comes last in the input sequence*.)

The expression `map:merge(($week, map{6:"Sonnabend"}), map{"duplicates":"use-first"})` returns `map{0:"Sonntag", 1:"Montag", 2:"Dienstag", 3:"Mittwoch", 4:"Donnerstag", 5:"Freitag", 6:"Samstag"}`. (*The value of the existing map is unchanged; the returned map contains all the entries from \$week, with one entry replaced by a new entry. Both input maps contain an entry with the key 6; the one used in the result is the one that comes first in the input sequence*.)

The expression `map:merge(($week, map{6:"Sonnabend"}), map{"duplicates":"combine"})` returns `map{0:"Sonntag", 1:"Montag", 2:"Dienstag", 3:"Mittwoch", 4:"Donnerstag", 5:"Freitag", 6:(("Samstag", "Sonnabend"))}`. (*The value of the existing map is unchanged; the returned map contains all the entries from \$week, with one entry replaced by a new entry. Both input maps contain an entry with the key 6; the entry that appears in the result is the sequence-concatenation of the entries in the input maps, retaining order*.)

21.2.3 [map:size](#)

Summary

Returns the number of entries in the supplied map.

Signature

```
map:size($map as map(*)) as xs:integer
```

Properties

This function is [deterministic^{FO30}](#), [context-independent^{FO30}](#), and [focus-independent^{FO30}](#).

Rules

The function `map:size` takes any [map](#) as its `$map` argument and returns the number of entries that are present in the map.

Examples

The expression `map:size(map{})` returns 0.

The expression `map:size(map{"true":1, "false":0})` returns 2.

21.2.4 [map:keys](#)

Summary

Returns a sequence containing all the keys present in a map

Signature

```
map:keys($map as map(*)) as xs:anyAtomicType*
```

Properties

This function is [deterministic^{FO30}](#), [context-independent^{FO30}](#), and [focus-independent^{FO30}](#).

Rules

The function `map:keys` takes any [map](#) as its `$map` argument and returns the keys that are present in the map as a sequence of atomic values, in [implementation-dependent](#) order.

The function is **non-deterministic with respect to ordering** (see [Section 1.7.4 Properties of functions^{FO31}](#)).

This means that two calls with the same argument are not guaranteed to produce the results in the same order.

Notes

The number of items in the result will be the same as the number of entries in the map, and the result sequence will contain no duplicate values.

Examples

The expression `map:keys(map{1:"yes", 2:"no"})` returns some permutation of (1,2). (*The result is in implementation-dependent order.*)

21.2.5 [map:contains](#)

Summary

Tests whether a supplied map contains an entry for a given key

Signature

```
map:contains($map as map(*),
             $key as xs:anyAtomicType) as xs:boolean
```

Properties

This function is [deterministic^{FO30}](#), [context-independent^{FO30}](#), and [focus-independent^{FO30}](#).

Rules

The function `map:contains` returns true if the `map` supplied as `$map` contains an entry with the [same key](#) as the supplied value of `$key`; otherwise it returns false.

Examples

```
let $week := map{0:"Sonntag", 1:"Montag", 2:"Dienstag",
                 3:"Mittwoch", 4:"Donnerstag", 5:"Freitag", 6:"Samstag"}
```

The expression `map:contains($week, 2)` returns `true()`.

The expression `map:contains($week, 9)` returns `false()`.

The expression `map:contains(map{}, "xyz")` returns `false()`.

The expression `map:contains(map{"xyz":23}, "xyz")` returns `true()`.

The expression `map:contains(map{"abc":23, "xyz":()}, "xyz")` returns `true()`.

21.2.6 `map:get`

Summary

Returns the value associated with a supplied key in a given map.

Signature

```
map:get($map as map(*),
        $key as xs:anyAtomicType) as item()*
```

Properties

This function is [deterministic^{FO30}](#), [context-independent^{FO30}](#), and [focus-independent^{FO30}](#).

Rules

The function `map:get` attempts to find an entry within the `map` supplied as `$map` that has the [same key](#) as the supplied value of `$key`. If there is such an entry, it returns the associated value; otherwise it returns an empty sequence.

Notes

A return value of `()` from `map:get` could indicate that the key is present in the map with an associated value of `()`, or it could indicate that the key is not present in the map. The two cases can be distinguished by calling `map:contains`.

Invoking the [map](#) as a function item has the same effect as calling `get`: that is, when `$map` is a map, the expression `$map($K)` is equivalent to `map:get($map, $K)`. Similarly, the expression `map:get(map:get($map, 'employee'), 'name'), 'first')` can be written as `$map('employee')('name')('first')`.

Examples

```
let $week := map{0:"Sonntag", 1:"Montag", 2:"Dienstag",
  3:"Mittwoch", 4:"Donnerstag", 5:"Freitag",
  6:"Samstag"}
```

The expression `map:get($week, 4)` returns "Donnerstag".

The expression `map:get($week, 9)` returns `()`. (*When the key is not present, the function returns an empty sequence.*)

The expression `map:get(map:entry(7,()), 7)` returns `()`. (*An empty sequence as the result can also signify that the key is present and the associated value is an empty sequence.*)

21.2.7 [map:put](#)

Summary

Returns a map containing all the contents of the supplied map, but with an additional entry, which replaces any existing entry for the same key.

Signature

```
map:put($map      as map(*),
        $key       as xs:anyAtomicType,
        $value     as item()* ) as map(*)
```

Properties

This function is [deterministic](#)^{FO30}, [context-independent](#)^{FO30}, and [focus-independent](#)^{FO30}.

Rules

The function `map:put` returns a [map](#) that contains all entries from the supplied `$map`, with the exception of any entry whose key is the [same key](#) as `$key`, together with a new entry whose key is `$key` and whose associated value is `$value`.

The effect of the function call `map:put($MAP, $KEY, $VALUE)` is equivalent to the result of the following steps:

1. `let $MAP2 := map:remove($MAP, $KEY)`

This returns a map in which all entries with the same key as `$KEY` have been removed.

2. Construct and return a map containing:

- a. All the entries (key/value pairs) in `$MAP2`, and
- b. The entry `map:entry($KEY, $VALUE)`

Notes

There is no requirement that the type of \$key and \$value be consistent with the types of any existing keys and values in the supplied map.

Examples

```
let $week := map{0:"Sonntag", 1:"Montag", 2:"Dienstag",
  3:"Mittwoch", 4:"Donnerstag", 5:"Freitag",
  6:"Samstag"}
```

The expression `map:put($week, 6, "Sonnabend")` returns `map{0:"Sonntag", 1:"Montag", 2:"Dienstag", 3:"Mittwoch", 4:"Donnerstag", 5:"Freitag", 6:"Sonnabend"}`.

The expression `map:put($week, -1, "Unbekannt")` returns `map{0:"Sonntag", 1:"Montag", 2:"Dienstag", 3:"Mittwoch", 4:"Donnerstag", 5:"Freitag", 6:"Samstag", -1:"Unbekannt"}`.

21.2.8 map:entry**Summary**

Returns a map that contains a single entry (a key-value pair).

Signature

```
map:entry($key    as xs:anyAtomicType,
          $value   as item()*) as map(*)
```

Properties

This function is [deterministic^{FO30}](#), [context-independent^{FO30}](#), and [focus-independent^{FO30}](#).

Rules

The function `map:entry` returns a [map](#) which contains a single entry. The key of the entry in the new map is `$key`, and its associated value is `$value`.

Notes

The function `map:entry` is intended primarily for use in conjunction with the function `map:merge`. For example, a map containing seven entries may be constructed like this:

```
map:merge((
  map:entry("Su", "Sunday"),
  map:entry("Mo", "Monday"),
  map:entry("Tu", "Tuesday"),
  map:entry("We", "Wednesday"),
  map:entry("Th", "Thursday"),
  map:entry("Fr", "Friday"),
  map:entry("Sa", "Saturday")
))
```

Unlike the map expression (`map{...}`), this technique can be used to construct a map with a variable number of entries, for example:

```
map:merge(for $b in //book return map:entry($b/isbn, $b))
```

Examples

The expression `map:entry("M", "Monday")` returns `{"M": "Monday"}`.

21.2.9 [map:remove](#)

Summary

Returns a map containing all the entries from a supplied map, except those having a specified key.

Signature

```
map:remove($map as map(*),
           $keys as xs:anyAtomicType*) as map(*)
```

Properties

This function is [deterministic^{FO30}](#), [context-independent^{FO30}](#), and [focus-independent^{FO30}](#).

Rules

The function `map:remove` returns a [map](#) containing all the entries in `$map` except for any entry whose key is the [same key](#) as an item in `$keys`.

No failure occurs if an item in `$keys` does not correspond to any entry in `$map`; that key value is simply ignored.

The effect of the function call `map:remove($MAP, $KEY)` can be described more formally as the result of the expression below:

```
map:merge (
  map:for-each (
    $MAP, function($k, $v) {
      if (some $key in $KEY satisfies op:same-key($k, $key))
        then ()
        else map:entry($k, $v)
    } ) )
```

Examples

```
let $week := map{0:"Sonntag", 1:"Montag", 2:"Dienstag",
  3:"Mittwoch", 4:"Donnerstag", 5:"Freitag", 6:"Samstag"}
```

The expression `map:remove($week, 4)` returns `map{0:"Sonntag", 1:"Montag",
 2:"Dienstag", 3:"Mittwoch", 5:"Freitag", 6:"Samstag"}`.

The expression `map:remove($week, 23)` returns `map{0:"Sonntag", 1:"Montag",
 2:"Dienstag", 3:"Mittwoch", 4:"Donnerstag", 5:"Freitag", 6:"Samstag"}`.

The expression `map:remove($week, (0, 6 to 7))` returns `map{1:"Montag", 2:"Dienstag",
 3:"Mittwoch", 4:"Donnerstag", 5:"Freitag"}`.

The expression `map:remove($week, ())` returns `>map{0:"Sonntag", 1:"Montag",
 2:"Dienstag", 3:"Mittwoch", 4:"Donnerstag", 5:"Freitag", 6:"Samstag"}`.

21.2.10 [map:for-each](#)

Summary

Applies a supplied function to every entry in a map, returning the concatenation of the results.

Signature

```
map:for-    $map      as map(*),
each(       $action   as function(xs:anyAtomicType, item()*)
            item()*) as item()*
```

Properties

This function is [deterministic](#)^{FO30}, [context-independent](#)^{FO30}, and [focus-independent](#)^{FO30}.

Rules

The function `map:for-each` takes any [map](#) as its `$map` argument and applies the supplied function to each entry in the map, in [implementation-dependent](#) order; the result is the sequence obtained by concatenating the results of these function calls.

The function is **non-deterministic with respect to ordering** (see [Section 1.7.4 Properties of functions](#)^{FO31}).

This means that two calls with the same arguments are not guaranteed to process the map entries in the same order.

The function supplied as `$action` takes two arguments. It is called supplying the key of the map entry as the first argument, and the associated value as the second argument.

Examples

The expression `map:for-each(map{1:"yes", 2:"no"}, function($k, $v){$k})` returns some permutation of (1,2). (*This function call is equivalent to calling `map:keys`. The result is in implementation-dependent order.*)

The expression `distinct-values(map:for-each(map{1:"yes", 2:"no"}, function($k, $v){$v}))` returns some permutation of ("yes", "no"). (*This function call returns the distinct values present in the map, in implementation-dependent order.*)

The expression `map:merge(map:for-each(map{"a":1, "b":2}, function($k, $v){map:entry($k, $v+1)}))` returns `map{"a":2, "b":3}`. (*This function call returns a map with the same keys as the input map, with the value of each entry increased by one.*)

Example: Converting a Map to an Element Node

This XQuery example converts the entries in a map to attributes on a newly constructed element node.

```
let
  $dimensions := map{'height': 3, 'width': 4, 'depth': 5};
return
<box>{
  map:for-each($dimensions, function ($k, $v) { attribute {$k} {$v} })
}</box>
```

The result is the element `<box height="3" width="4" depth="5"/>`.

21.2.11 `map:find`

Summary

Searches the supplied input sequence and any contained maps and arrays for a map entry with the supplied key, and returns the corresponding values.

Signature

```
map:find($input as item()*,
          $key    as xs:anyAtomicType) as array(*)
```

Properties

This function is [deterministic^{FO30}](#), [context-independent^{FO30}](#), and [focus-independent^{FO30}](#).

Rules

The function `map:find` searches the sequence supplied as `$input` looking for map entries whose key is the [same key](#) as `$key`. The associated value in any such map entry (each being in general a sequence) is returned as a member of the result array.

The search processes the `$input` sequence using the following recursively-defined rules (any equivalent algorithm may be used provided it delivers the same result, respecting those rules that constrain the order of the result):

1. To process a sequence, process each of its items in order.
2. To process an item that is an array, process each of the array's members in order (each member is, in general, a sequence).
3. To process an item that is a map, then for each key-value entry (K, V) in the map (in implementation-dependent order) perform both of the following steps, in order:
 - a. If K is the same key as $\$key$, then add V as a new member to the end of the result array.
 - b. Process V (which is, in general, a sequence).
4. To process an item that is neither a map nor an array, do nothing. (Such items are ignored).

Notes

If $\$input$ is an empty sequence, map, or array, or if the requested $\$key$ is not found, the result will be a zero-length array.

Examples

```
let $responses := [map{0:'no', 1:'yes'},
                  map{0:'non', 1:'oui'},
                  map{0:'nein', 1:('ja', 'doch')}]
```

The expression `map:find($responses, 0)` returns `['no', 'non', 'nein']`.

The expression `map:find($responses, 1)` returns `['yes', 'oui', ('ja', 'doch')]`.

The expression `map:find($responses, 2)` returns `[]`.

```
let $inventory := map{"name":"car", "id":"QZ123",
                      "parts": [map{name:"engine", "id":"YW678", "parts":[]}]}
```

The expression `map:find($inventory, "parts")` returns `[[map{name:"engine", "id":"YW678", "parts":[]}], []]`.

21.2.12 [fn:collation-key](#)

Summary

Given a string value and a collation, generates an internal value called a collation key, with the property that the matching and ordering of collation keys reflects the matching and ordering of strings under the specified collation.

Signatures

```
fn:collation-key($key as xs:string) as xs:base64Binary
```

```
fn:collation-key($key          as xs:string,
                  $collation as xs:string) as xs:base64Binary
```

Properties

This function is deterministic^{FO30}, context-dependent^{FO30}, and focus-independent^{FO30}. It depends on collations.

Rules

Calling the one-argument version of this function is equivalent to calling the two-argument version supplying the default collation as the second argument.

The function returns an [implementation-dependent](#) value with the property that, for any two strings \$K1 and \$K2:

- `collation-key($K1, $C) eq collation-key($K2, $C)` if and only if `compare($K1, $K2, $C) eq 0`
- `collation-key($K1, $C) lt collation-key($K2, $C)` if and only if `compare($K1, $K2, $C) lt 0`

The collation used by this function is determined in the same way as for other functions accepting a collation URI argument. Collation keys are defined as `xs:base64Binary` values to ensure unambiguous and context-free comparison semantics.

An implementation is free to generate a collation key in any convenient way provided that it always generates the same collation key for two strings that are equal under the collation, and different collation keys for strings that are not equal. This holds only within a single [execution scope](#)^{FO30}; an implementation is under no obligation to generate the same collation keys during a subsequent unrelated query or transformation.

It is possible to define collations that do not have the ability to generate collation keys. Supplying such a collation will cause the function to fail. The ability to generate collation keys is an [implementation-defined](#) property of the collation.

Error Conditions

An error is raised [\[ERR FOCH0004\]](#)^{FO31} if the specified collation does not support the generation of collation keys.

Notes

The function is provided primarily for use with maps. If a map is required where codepoint equality is inappropriate for comparing keys, then a common technique is to normalize the key so that equality matching becomes feasible. There are many ways keys can be normalized, for example by use of functions such as `fn:upper-case`, `fn:lower-case`, `fn:normalize-space`, or `fn:normalize-unicode`, but this function provides a way of normalizing them according to the rules of a specified collation. For example, if the collation ignores accents, then the function will generate the same collation key for two input strings that differ only in their use of accents.

The result of the function is defined to be an `xs:base64Binary` value. Binary values are chosen because they have unambiguous and context-free comparison semantics, because the value space is unbounded, and because the ordering rules are such that between any two values in the ordered value space, an arbitrary number of further values can be interpolated. The choice between `xs:base64Binary` and `xs:hexBinary` is arbitrary; the only operation that behaves differently between the two binary data types is conversion to/from a string, and this operation is not one that is normally required for effective use of collation keys.

For collations based on the Unicode Collation Algorithm, an algorithm for computing collation keys is provided in [\[UNICODE TR10\]](#). Implementations are NOT REQUIRED to use this algorithm.

This specification does not mandate that collation keys should retain ordering. This is partly because the primary use case is for maps, where only equality comparisons are required, and partly to allow the use of binary data types (which are currently unordered types) for the result. The specification may be revised in a future release to specify that ordering is preserved.

The fact that collation keys are ordered can be exploited in XQuery, whose `order by` clause does not allow the collation to be selected dynamically. This restriction can be circumvented by rewriting the clause `order by $e/@key collation "URI"` as `order by fn:collation-key($e/@key, $collation)`, where `$collation` allows the collation to be chosen dynamically.

Note that `xs:base64Binary` becomes an ordered type in XPath 3.1, making binary collation keys possible. In an implementation that adheres strictly to XPath 3.0, collation keys can be used only for equality matching, not for ordering operations.

Examples

```
let $C := 'http://www.w3.org/2013/collation/UCA?strength=primary'
```

The expression `map:merge((map{collation-key("A", $C):1}, map{collation-key("a", $C):2}), map{"duplicates":"use-last"})(collation-key("A", $C))` returns 2. (*Given that the keys of the two entries are equal under the rules of the chosen collation, only one of the entries can appear in the result; the one that is chosen is the one from the last map in the input sequence.*)

The expression `let $M := map{collation-key("A", $C):1, collation-key("B", $C):2} return $M(collation-key("a", $C))` returns 1. (*The strings "A" and "a" have the same collation key under this collation.*)

As the above examples illustrate, it is important that when the `collation-key` function is used to add entries to a map, then it must also be used when retrieving entries from the map. This process can be made less error-prone by encapsulating the map within a function: `function($k) {$M(collation-key($k, $collation))}`.

21.2.13 `fn:deep-equal`

The `deep-equal`^{FO30} function, when used with XSLT 3.0, is defined to handle maps in the way that is defined in the [\[Functions and Operators 3.1\]](#) specification of the function.

Specifically, two maps are deep-equal if they have the same number of entries, and if there is a one-to-one correspondence in the sense that for every entry E_1 in the first map, there is a corresponding entry E_2 in the second map, such that the keys of E_1 and E_2 are equal under the `op:same-key` relation (see [21.2.1 op:same-key](#)), and the corresponding values are equal under the `deep-equal` relation, invoked recursively using the collation supplied as argument to the original `deep-equal` call, or its default. Note that collations are not used for comparing keys.

21.3 Map Instructions

Two instructions are added to XSLT to facilitate the construction of maps.

```
<!-- Category: instruction -->
<xsl:map>
  <!-- Content: sequence-constructor -->
</xsl:map>
```

The instruction [xsl:map](#) constructs and returns a new map.

The contained sequence constructor MUST evaluate to a sequence of maps: call this \$maps.

The result of the instruction (other than the choice of error code) is then given by the XPath 3.1 expression:

```
map:merge($maps, map{"duplicates":"reject"})
```

Note:

Informally: if there are duplicate keys among the sequence of maps, a dynamic error occurs. Otherwise, the resulting map contains the union of the map entries from the supplied sequence of maps.

[ERR XTDE3365] A [dynamic error](#) occurs if the set of keys in the maps resulting from evaluating the sequence constructor contains duplicates.

There is no requirement that the supplied input maps should have the same or compatible types. The type of a map (for example `map(xs:integer, xs:string)`) is descriptive of the entries it currently contains, but is not a constraint on how the map may be combined with other maps.

[ERR XTTE3375] A type error occurs if the result of evaluating the sequence constructor is not an instance of the required type `map(*)`.

Note:

In practice, the effect of this rule is that the sequence constructor contained in the [xsl:map](#) instruction is severely constrained: it doesn't make sense, for example, for it to contain instructions such as [xsl:element](#) that create new nodes. As with other type errors, processors are free to signal the error statically if they are able to determine that the sequence constructor would always fail when evaluated.

```
<!-- Category: instruction -->
<xsl:map-entry>
  key = expression
  select? = expression >
  <!-- Content: sequence-constructor -->
</xsl:map-entry>
```

The instruction [xsl:map-entry](#) constructs and returns a singleton map: that is, a map which contains one key and one value. Such a map is primarily used as a building block when constructing maps using the [xsl:map](#) instruction.

The `select` attribute and the contained sequence constructor are mutually exclusive: if a `select` attribute is present, then the content **MUST** be empty except optionally for `xsl:fallback` instructions.

[ERR XTSE3280] It is a [static error](#) if the `select` attribute of the `xsl:map-entry` element is present unless the element has no children other than `xsl:fallback` elements.

The key of the entry in the new map is the value obtained by evaluating the expression in the `key` attribute, converted to the required type `xs:anyAtomicType` by applying the [function conversion rules](#). If the supplied key (after conversion) is of type `xs:untypedAtomic`, it is cast to `xs:string`.

The associated value is the value obtained by evaluating the expression in the `select` attribute, or the contained sequence constructor, with no conversion. If there is no `select` attribute and the sequence constructor is empty, the associated value is the empty sequence.

Example: Using XSLT instructions to create a fixed map

The following example binds a variable to a map whose content is statically known:

```
<xsl:variable name="week" as="map(xs:string, xs:string)">
  <xsl:map>
    <xsl:map-entry key="'Mo'" select="'Monday'"/>
    <xsl:map-entry key="'Tu'" select="'Tuesday'"/>
    <xsl:map-entry key="'We'" select="'Wednesday'"/>
    <xsl:map-entry key="'Th'" select="'Thursday'"/>
    <xsl:map-entry key="'Fr'" select="'Friday'"/>
    <xsl:map-entry key="'Sa'" select="'Saturday'"/>
    <xsl:map-entry key="'Su'" select="'Sunday'"/>
  </xsl:map>
</xsl:variable>
```

Example: Using XSLT instructions to create a computed map

The following example binds a variable to a map acting as an index into a source document:

```
<xsl:variable name="index" as="map(xs:string, element(employee))">
  <xsl:map>
    <xsl:for-each select="//employee">
      <xsl:map-entry key="@empNr" select=". "/>
    </xsl:for-each>
  </xsl:map>
</xsl:variable>
```

21.4 Map Constructors

A Map Constructor is a new kind of expression added to the syntax of XPath.

Note:

Map Constructors are defined in XPath 3.1. They are available in XSLT 3.0 whether or not XPath 3.1 is supported. The specification given here is intended to be identical to the specification in XPath 3.1.

The syntax of [PrimaryExpr^{XP30}](#) is extended to permit `MapConstructor` as an additional alternative.

MapConstructor

[52] PrimaryExpr	::= Literal VarRef ParenthesizedExpr ContextItemExpr FunctionCall FunctionItemExpr MapConstructor
[202] MapConstructor	::= "map" "{" (MapConstructorEntry (", " MapConstructorEntry)*)? "}"
[203] MapConstructorEntry	::= MapKeyExpr ":" MapValueExpr
[204] MapKeyExpr	::= ExprSingle^{XP30}
[205] MapValueExpr	::= ExprSingle^{XP30}

Note:

In some circumstances, it is necessary to include whitespace before or after the colon to ensure that this grammar is correctly parsed; this arises for example when the `KeyExpr` ends with a name and the `ValueExpr` starts with a name.

The value of the expression is a map whose entries correspond to the key-value pairs obtained by evaluating the successive `KeyExpr` and `ValueExpr` expressions.

Each `KeyExpr` expression is evaluated and atomized; a type error [\[ERR_XPTY0004\]^{XP30}](#) occurs if the result is not a single atomic value. If the key is of type `xs:untypedAtomic` it is converted to `xs:string`. The associated value is the result of evaluating the corresponding `ValueExpr`. If two or more entries have the [same key](#) then a dynamic error occurs [see [ERR_XTDE3365](#)].

For example, the following expression constructs a map with seven entries:

```
map {
  "Su" : "Sunday",
  "Mo" : "Monday",
  "Tu" : "Tuesday",
  "We" : "Wednesday",
  "Th" : "Thursday",
  "Fr" : "Friday",
  "Sa" : "Saturday"
}
```

Note:

Unlike the [map:merge](#) function, the number of entries in a map that is constructed using a map expression is known statically.

21.5 The Map Lookup Operator

A new operator is introduced into XPath to allow convenient lookup of entries in a map (or a sequence of maps), knowing the key.

Note:

Map Lookup Expressions are defined in XPath 3.1. They are available in XSLT 3.0 whether or not XPath 3.1 is supported. The specification given here is intended to be identical to the specification in XPath 3.1.

The operator is available in two forms: as a unary (prefix) operator, and as a postfix operator.

21.5.1 The Unary Lookup Operator

```
[76] UnaryLookup ::= "?" KeySpecifier [54] KeySpecifier ::= NCName | IntegerLiteral | ParenthesizedExpr | "*"
```

A UnaryLookup expression returns a sequence of values selected from the map that is the context item. If the context item is not a map or an array, a type error is raised [err:XPTY0004]

The semantics are as follows:

1. If the KeySpecifier is an NCName, the UnaryLookup expression ?KS is equivalent to .("KS"). For example, \$emp[?name='Jim'] is shorthand for \$emp[.(name)='Jim'].
2. If the KeySpecifier is an IntegerLiteral, the UnaryLookup expression ?N is equivalent to .(N). This form is only useful for maps whose keys are numeric. For example, \$temp[?7 > 30] is shorthand for \$temp[.(7) > 30].
3. If the KeySpecifier is a ParenthesizedExpr, the UnaryLookup expression ?(EXP) is equivalent to .(EXP). This form allows arbitrary keys, including keys computed dynamically. For example, \$emp[?('Year of Birth') > 1980] is shorthand for \$emp[.(Year of Birth) > 1980]
4. If the KeySpecifier is a wildcard ("*") the UnaryLookup expression ?* is equivalent to the expression for \$k in map:keys(.) return .(\$k). That is, it returns the sequence-concatenation of all the values in the map; since the order of keys is implementation-dependent, so is the order of these values.

21.5.2 The Postfix Lookup Operator

```
[53] Lookup ::= "?" KeySpecifier
```

The semantics of the postfix lookup operator are defined in terms of the unary lookup operator. The left-hand operand must be a map, or a sequence of maps. The KeySpecifier is applied to each of these maps, in order, and the

results are sequence-concatenated.

1. If the **KeySpecifier** is an NCName, the Postfix Lookup expression $E?KS$ is equivalent to $E!?KS$. For example, if $\$emps$ is a sequence of maps containing information about employees, then $\$emps?name$ selects the names of the employees: it is equivalent to $\$emps!map:get(., "name")$.
2. If the **KeySpecifier** is an **IntegerLiteral**, the Postfix Lookup expression $E?N$ is equivalent to $E!?N$. This form is only useful for maps whose keys are numeric.
For example, if $\$emps$ is a sequence of maps containing information about employees, and if one of the entries in this map represents the salary history, as a map whose keys are the relevant year and whose associated value is the salary, then $\$emps[?name='John']?2012$ returns the value of John's salary in the year 2012.
3. If the **KeySpecifier** is a **ParenthesizedExpr**, the Postfix Lookup expression $E?(EXP)$ is equivalent to `for $m in E, $k in EXP return $m!?($k)`. This form allows arbitrary keys, including keys computed dynamically. It also allows multiple keys.
For example, if $\$emps$ is a sequence of maps containing information about employees, and if one of the entries in this map represents the salary history, as a map whose keys are the relevant year and whose associated value is the salary, then $\$emps[?name='John']?(2012 to 2015)$ returns the value of John's salary in the years 2012 through 2015.
4. If the **KeySpecifier** is a wildcard ("*") the Postfix Lookup expression $E?*$ is equivalent to the expression $E!?*$. That is, it returns the sequence-concatenation of all the values in all the maps; since the order of keys within each map is implementation-dependent, so is the order of these values.

[21.6 Maps and Streaming](#)

Maps have many uses, but their introduction to XSLT 3.0 was strongly motivated by streaming use cases. In essence, when a source document is processed in streaming mode, data that is encountered in the course of processing may need to be retained in variables for subsequent use, because the nodes cannot be revisited. This creates a need for a flexible data structure to accommodate such temporary data, and maps were designed to fulfil this need.

The entries in a map are not allowed to contain references to streamed nodes. This is achieved by ensuring that for all constructs that supply content to be included in a map (for example the third argument of map:put, and the select attribute of xsl:map-entry), the relevant operand is defined to have operand usage navigation. Because maps cannot contain references to streamed nodes, they are effectively grounded, and can therefore be used freely in contexts (such as parameters to functions or templates) where only grounded operands are permitted.

The xsl:map instruction, and the XPath MapConstructor construct, are exceptions to the general rule that during streaming, only one downward selection (one consuming subexpression) is permitted. They share this characteristic with xsl:fork. As with xsl:fork, a streaming processor is expected to be able to construct the map during a single pass of the streamed input document, which may require multiple expressions to be evaluated in parallel.

In the case of the xsl:map instruction, this exemption applies only in the case where the instruction consists exclusively of xsl:map-entry (and xsl:fallback) children, and not in more complex cases where the map entries are constructed dynamically (for example using a control flow implemented using xsl:choose, xsl:for-each, or xsl:call-template). Such cases may, of course, be streamable if they only have a single consuming subexpression.

For example, the following XPath expression is streamable, despite making two downward selections:

```
let $m := map{'price':xs:decimal(price), 'discount':xs:decimal(discount)}
return ($m?price - $m?discount)
```

Analysis:

1. Because the `return` clause is motionless, the sweep of the `let` expression is the sweep of the map expression (the expression in curly brackets).
2. The sweep of a map expression is the maximum sweep of its key/value pairs.
3. For both key/value pairs, the key is motionless and the value is consuming.
4. The expression carefully atomizes both values, because retaining references to streamed nodes in a map is not permitted.
5. Therefore the map expression, and hence the expression as a whole, is grounded and consuming.

See also: [19.8.8.17 Streamability of Map Constructors](#), [19.8.4.23 Streamability of `xsl:map`](#), [19.8.4.24 Streamability of `xsl:map-entry`](#)

[21.7 Examples using Maps](#)

This section gives some examples of where maps can be useful.

Example: Using Maps with `xsl:iterate`

This example uses maps in conjunction with the `xsl:iterate` instruction to find the highest-earning employee in each department, in a single streaming pass of an input document containing employee records.

```
<xsl:source-document streamable="yes" href="employees.xml">
  <xsl:iterate select="*/employee">
    <xsl:param name="highest-earners"
      as="map(xs:string, element(employee))"
      select="map{}"/>
    <xsl:on-completion>
      <xsl:for-each select="map:keys($highest-earners)">
        <department name=". ">
          <xsl:copy-of select="$highest-earners(.)"/>
        </department>
      </xsl:for-each>
    </xsl:on-completion>
    <xsl:variable name="this" select="copy-of(.)" as="element(employee)"/>
    <xsl:next-iteration>
      <xsl:with-param name="highest-earners"
        select="let $existing := $highest-earners($this/department)
                return if ($existing/salary gt $this/salary)
                        then $highest-earners
                        else map:put($highest-earners, $this/department,
$this)"/>
    </xsl:next-iteration>
  </xsl:iterate>
</xsl:source-document>
```

Example: Using Maps to Implement Complex Numbers

A complex number might be represented as a map with two entries, the keys being the `xs:boolean` value `true` for the real part, and the `xs:boolean` value `false` for the imaginary part. A library for manipulation of complex numbers might include functions such as the following:

```

<xsl:variable name="REAL" static="yes" as="xs:int" select="0"/>
<xsl:variable name="IMAG" static="yes" as="xs:int" select="1"/>

<xsl:function name="i:complex" as="map(xs:int, xs:double)">
  <xsl:param name="real" as="xs:double"/>
  <xsl:param name="imaginary" as="xs:double"/>
  <xsl:sequence select="map{ $REAL : $real, $IMAG : $imaginary }"/>
</xsl:function>

<xsl:function name="i:real" as="xs:double">
  <xsl:param name="complex" as="map(xs:int, xs:double)" />
  <xsl:sequence select="$complex($REAL)" />
</xsl:function>

<xsl:function name="i:imaginary" as="xs:double">
  <xsl:param name="complex" as="map(xs:int, xs:double)" />
  <xsl:sequence select="$complex($IMAG)" />
</xsl:function>

<xsl:function name="i:add" as="map(xs:int, xs:double)">
  <xsl:param name="arg1" as="map(xs:int, xs:double)" />
  <xsl:param name="arg2" as="map(xs:int, xs:double)" />
  <xsl:sequence select="i:complex(i:real($arg1)+i:real($arg2),
                                 i:imaginary($arg1)+i:imaginary($arg2))"/>
</xsl:function>

<xsl:function name="i:multiply" as="map(xs:boolean, xs:double)">
  <xsl:param name="arg1" as="map(xs:boolean, xs:double)" />
  <xsl:param name="arg2" as="map(xs:boolean, xs:double)" />
  <xsl:sequence select="i:complex(
    i:real($arg1)*i:real($arg2) - i:imaginary($arg1)*i:imaginary($arg2),
    i:real($arg1)*i:imaginary($arg2) + i:imaginary($arg1)*i:real($arg2))"/>
</xsl:function>
```

Example: Using a Map as an Index

Given a set of book elements, it is possible to construct an index in the form of a map allowing the books to be retrieved by ISBN number.

Assume the book elements have the form:

```
<book>
  <isbn>0470192747</isbn>
  <author>Michael H. Kay</author>
  <publisher>Wiley</publisher>
  <title>XSLT 2.0 and XPath 2.0 Programmer's Reference</title>
</book>
```

An index may be constructed as follows:

```
<xsl:variable name="isbn-index" as="map(xs:string, element(book))"
  select="map:merge(for $b in //book return map{$b/isbn : $b})"/>
```

This index may then be used to retrieve the book for a given ISBN using either of the expressions `map:get($isbn-index, "0470192747")` or `$isbn-index("0470192747")`.

In this simple form, this replicates the functionality available using [xsl:key](#) and the [key](#) function. However, it also provides capabilities not directly available using the [key](#) function: for example, the index can include book elements in multiple source documents. It also allows processing of all the books using a construct such as `<xsl:for-each select="map:keys($isbn-index)">`

Example: A Map containing Named Functions

As in JavaScript, a map whose keys are strings and whose associated values are function items can be used in a similar way to a class in object-oriented programming languages.

Suppose an application needs to handle customer order information that may arrive in three different formats, with different hierachic arrangements:

1. Flat structure:

```
<customer id="c123">...</customer>
<product id="p789">...</product>
<order customer="c123" product="p789">...</order>
```

2. Orders within customer elements:

```
<customer id="c123">
  <order product="p789">...</order>
</customer>
<product id="p789">...</product>
```

3. Orders within product elements:

```
<customer id="c123">...</customer>
<product id="p789">
  <order customer="c123">...</order>
</product>
```

An application can isolate itself from these differences by defining a set of functions to navigate the relationships between customers, orders, and products: `orders-for-customer`, `orders-for-product`, `customer-for-order`, `product-for-order`. These functions can be implemented in different ways for the three different input formats. For example, with the first format the implementation might be:

```
<xsl:variable name="flat-input-functions" as="map(xs:string, function(*))*
  select="map{
    'orders-for-customer' :
      function($c as element(customer)) as element(order)*
        {$c/../order[@customer=$c/@id]},
    'orders-for-product' :
      function($p as element(product)) as element(order)*
        {$p/../order[@product=$p/@id]},
    'customer-for-order' :
      function($o as element(order)) as element(customer)
        {$o/../customer[@id=$o/@customer]},
    'product-for-order' :
      function($o as element(order)) as element(product)
        {$o/../product[@id=$o/@product]} }
```

Having established which input format is in use, the application can bind the appropriate implementation of these functions to a variable such as `$input-navigator`, and can then process the input using XPath expressions such as the following, which selects all products for which there is no order:

```
//product[empty($input-navigator("orders-for-product"))(.)]]
```

22 Processing JSON Data

JSON is a popular format for exchange of structured data on the web: it is specified in [\[RFC 7159\]](#). This section describes facilities allowing JSON data to be processed using XSLT.

Note:

RFC7159 is taken as the definitive specification of JSON for the purposes of this document. The RFC explains its relationship with other JSON specifications such as [\[ECMA-404\]](#).

Note:

XPath 3.1 incorporates the functions defined in this section. It also provides additional JSON capability, in the form of functions `parse-json`, `json-doc`, and extensions to the `serialize`^{FO30} function. These facilities are incorporated in XSLT 3.0 only if the XPath 3.1 feature is supported. They depend on support for arrays.

22.1 XML Representation of JSON

This specification defines a mapping from JSON data to XML (specifically, to XDM instances). A function `json-to-xml` is provided to take a JSON string as input and convert it to the XML representation. Two stylesheet modules are provided to perform the reverse transformation: one produces JSON in compact linear form, the other in indented form suitable for display, editing, or printing.

The XML representation is designed to be capable of representing any valid JSON text other than one that uses characters which are not valid in XML. The transformation is lossless: that is, distinct JSON texts convert to distinct XML representations. When converting JSON to XML, options are provided to reject unsupported characters, to replace them with a substitute character, or to leave them in backslash-escaped form.

The following example demonstrates the correspondence of a JSON text and the corresponding XML representation.

Example: A JSON Text and its XML Representation

Consider the following JSON text:

```
{  
    "desc"      : "Distances between several cities, in kilometers.",  
    "updated"   : "2014-02-04T18:50:45",  
    "uptodate": true,  
    "author"   : null,  
    "cities"   : {  
        "Brussels": [  
            {"to": "London",     "distance": 322},  
            {"to": "Paris",      "distance": 265},  
            {"to": "Amsterdam", "distance": 173}  
        ],  
        "London": [  
            {"to": "Brussels",   "distance": 322},  
            {"to": "Paris",      "distance": 344},  
            {"to": "Amsterdam", "distance": 358}  
        ],  
        "Paris": [  
            {"to": "Brussels",   "distance": 265},  
            {"to": "London",     "distance": 344},  
            {"to": "Amsterdam", "distance": 431}  
        ],  
        "Amsterdam": [  
            {"to": "Brussels",   "distance": 173},  
            {"to": "London",     "distance": 358},  
            {"to": "Paris",      "distance": 431}  
        ]  
    }  
}
```

The XML representation of this text is as follows. Whitespace is included in the XML representation for purposes of illustration, and is ignored by the stylesheets that convert XML to JSON, but it will not be present in the output of the [json-to-xml](#) function.

```
<map xmlns="http://www.w3.org/2005/xpath-functions">
    <string key='desc'>Distances between several cities, in kilometers.
</string>
    <string key='updated'>2014-02-04T18:50:45</string>
    <boolean key="uptodate">true</boolean>
    <null key="author"/>
    <map key='cities'>
        <array key="Brussels">
            <map>
                <string key="to">London</string>
                <number key="distance">322</number>
            </map>
            <map>
                <string key="to">Paris</string>
                <number key="distance">265</number>
            </map>
            <map>
                <string key="to">Amsterdam</string>
                <number key="distance">173</number>
            </map>
        </array>
        <array key="London">
            <map>
                <string key="to">Brussels</string>
                <number key="distance">322</number>
            </map>
            <map>
                <string key="to">Paris</string>
                <number key="distance">344</number>
            </map>
            <map>
                <string key="to">Amsterdam</string>
                <number key="distance">358</number>
            </map>
        </array>
        <array key="Paris">
            <map>
                <string key="to">Brussels</string>
                <number key="distance">265</number>
            </map>
            <map>
                <string key="to">London</string>
                <number key="distance">344</number>
            </map>
            <map>
                <string key="to">Amsterdam</string>
                <number key="distance">431</number>
            </map>
        </array>
        <array key="Amsterdam">
            <map>
                <string key="to">Brussels</string>
                <number key="distance">173</number>
            </map>
        </array>
    </map>
```

```

<map>
  <string key="to">London</string>
  <number key="distance">358</number>
</map>
<map>
  <string key="to">Paris</string>
  <number key="distance">431</number>
</map>
</array>
</map>
</map>

```

An XSD 1.0 schema for the XML representation is provided in [B.1 Schema for the XML Representation of JSON](#). It is not necessary to import this schema (using `xsl:import-schema`) unless the stylesheet makes explicit reference to the components defined in the schema. If the stylesheet does import a schema for the namespace `http://www.w3.org/2005/xpath-functions`, then:

1. The processor (if it is schema-aware) **MUST** recognize an `xsl:import-schema` declaration for this namespace, whether or not the `schema-location` is supplied.
2. If a `schema-location` is provided, then the schema document at that location **MUST** be equivalent to the schema document at [B.1 Schema for the XML Representation of JSON](#); the effect if it is not is [implementation-dependent](#)

The rules governing the mapping from JSON to XML are as follows. In these rules, the phrase “an element named N” is to be interpreted as meaning “an element node whose local name is N and whose namespace URI is `http://www.w3.org/2005/xpath-functions`”.

1. The JSON value `null` is represented by an element named `null`, with empty content.
2. The JSON values `true` and `false` are represented by an element named `boolean`, with content conforming to the type `xs:boolean`.
3. A JSON number is represented by an element named `number`, with content conforming to the type `xs:double`, with the additional restriction that the value must not be positive or negative infinity, nor NaN.
4. A JSON string is represented by an element named `string`, with content conforming to the type `xs:string`.
5. A JSON array is represented by an element named `array`. The content is a sequence of child elements representing the members of the array in order, each such element being the representation of the array member obtained by applying these rules recursively.
6. A JSON object is represented by an element named `map`. The content is a sequence of child elements each of which represents one of the name/value pairs in the object. The representation of the name/value pair `N:V` is obtained by taking the element that represents the value `V` (by applying these rules recursively) and adding an attribute with name `key` (in no namespace), whose value is `N` as an instance of `xs:string`.

The attribute `escaped="true"` may be specified on a `string` element to indicate that the string value contains backslash-escaped characters that are to be interpreted according to the JSON rules. The attribute `escaped-key="true"` may be specified on any element with a `key` attribute to indicate that the key contains backslash-escaped characters that are to be interpreted according to the JSON rules. Both attributes have the default value `false`.

The JSON grammar for `number` is a subset of the lexical space of the XSD type `xs:double`. The mapping from JSON `number` values to `xs:double` values is defined by the XPath rules for casting from `xs:string` to

`xs:double`. Note that these rules will never generate an error for out-of-range values; instead very large or very small values will be converted to `+INF` or `-INF`. Since JSON does not impose limits on the range or precision of numbers, the conversion is not guaranteed to be lossless.

Although the order of entries in a JSON object is generally considered to have no significance, the function `json-to-xml` and the stylesheets that perform the reverse transformation both retain order.

The XDM representation of a JSON value may either be untyped (all elements annotated as `xs:untyped`, attributes as `xs:untypedAtomic`), or it may be typed. If it is typed, then it **MUST** have the type annotations obtained by validating the untyped representation against the schema given in [B.1 Schema for the XML Representation of JSON](#). If it is untyped, then it **MUST** be an XDM instance such that validation against this schema would succeed.

22.2 Option Parameter Conventions

This section describes conventions which in principle can be adopted by the specification of any function. At the time of writing, the function which invoke these conventions are `xml-to-json` and `json-to-xml`.

As a matter of convention, a number of functions defined in this document take a parameter whose value is a map, defining options controlling the detail of how the function is evaluated. Maps are a new data type introduced in XSLT 3.0.

For example, the function `fn:xml-to-json` has an options parameter allowing specification of whether the output is to be indented. A call might be written:

```
fn:xml-to-json($input, map{'indent':true()})
```

[**DEFINITION:** Functions that take an options parameter adopt common conventions on how the options are used. These are referred to as the **option parameter conventions**. These rules apply only to functions that explicitly refer to them.]

Where a function adopts the [option parameter conventions](#), the following rules apply:

1. The value of the relevant argument must be a map. The entries in the map are referred to as options: the key of the entry is called the option name, and the associated value is the option value. Option names defined in this specification are always strings (single `xs:string` values). Option values may be of any type.
2. The type of the options parameter in the function signature is always given as `map(*)`.
3. Although option names are described above as strings, the actual key may be any value that compares equal to the required string (using the `eq` operator with Unicode codepoint collation). For example, instances of `xs:untypedAtomic` or `xs:anyURI` are equally acceptable.

Note:

This means that the implementation of the function can check for the presence and value of particular options using the functions `map:contains` and/or `map:get`.

4. It is not an error if the options map contains options with names other than those described in this specification. Implementations **MAY** attach an [implementation-defined](#) meaning to such entries, and **MAY** define errors that arise if such entries are present with invalid values. Implementations **MUST** ignore such entries

unless they have a specific [implementation-defined](#) meaning. Implementations that define additional options in this way SHOULD use values of type `xs:QName` as the option names, using an appropriate namespace.

5. All entries in the options map are optional, and supplying an empty map has the same effect as omitting the relevant argument in the function call, assuming this is permitted.
6. For each named option, the function specification defines a required type for the option value. The value that is actually supplied in the map is converted to this required type using the [function conversion rules](#)^{XP31}. A type error [\[ERR_XPTY0004\]](#)^{XP30} occurs if conversion of the supplied value to the required type is not possible, or if this conversion delivers a coerced function whose invocation fails with a type error. A dynamic error occurs if the supplied value after conversion is not one of the permitted values for the option in question: the error codes for this error are defined in the specification of each function.

Note:

It is the responsibility of each function implementation to invoke this conversion; it does not happen automatically as a consequence of the function calling rules.

7. In cases where an option is list-valued, by convention the value may be supplied either as a sequence or as an array. Accepting a sequence is convenient if the value is generated programmatically using an XPath expression; while accepting an array allows the options to be held in an external file in JSON format, to be read using a call on the `fn:json-doc` function.
8. In cases where the value of an option is itself a map, the specification of the particular function must indicate whether or not these rules apply recursively to the contents of that map.

22.3 `fn:json-to-xml`

Summary

Parses a string supplied in the form of a JSON text, returning the results in the form of an XML document node.

Signatures

```
fn:json-to-xml($json-text as xs:string) as document-node()
```

```
fn:json-to-xml($json-text as xs:string,
               $options   as map(*)) as document-node()
```

Properties

This function is [nondeterministic](#)^{FO30}, [context-dependent](#)^{FO30}, and [focus-independent](#)^{FO30}. It depends on static base URI.

Rules

The effect of the one-argument form of this function is the same as calling the two-argument form with an empty map as the value of the `$options` argument.

The first argument is a JSON-text (see below) in the form of a string. The function parses this string to return an XDM node.

The `$options` argument can be used to control the way in which the parsing takes place. The value of the argument is a map. The options defined in this specification have keys that are strings. The effect of any map entries whose keys are not defined in this specification is implementation-defined; implementation-defined options SHOULD use QNames as keys. Implementations MUST ignore any entries in the map whose keys are not defined in this specification, unless the key has a specific [implementation-defined](#) meaning.

The entries that may appear in the `$options` map are as follows. The keys are `xs:string` values:

Key	Value	Meaning
<code>liberal</code>	Determines whether deviations from the syntax of RFC7159 are permitted. The value MUST be a boolean.	
<code>false</code>	The input MUST consist of an optional byte order mark (which is ignored) followed by a string that conforms to the grammar of JSON-text in [RFC 7159] . An error MUST be raised (see below) if the input does not conform to the grammar.	
<code>true</code>	The input MAY contain deviations from the grammar of [RFC 7159] , which are handled in an implementation-defined way. (Note: some popular extensions include allowing quotes on keys to be omitted, allowing a comma to appear after the last item in an array, allowing leading zeroes in numbers, and allowing control characters such as tab and newline to be present in unescaped form.) Since the extensions accepted are implementation-defined, an error MAY be raised (see below) if the input does not conform to the grammar.	
<code>validate</code>	If the <code>\$options</code> map contains an entry with the key "validate", then the value MUST be an <code>xs:boolean</code> . The default is <code>true</code> for a schema-aware processor, <code>false</code> for a non-schema-aware processor. If the value <code>true</code> is supplied and the processor is not schema-aware, a dynamic error results [see ERR_XTDE3245]. It is not necessary that the containing stylesheet should import the relevant schema.	
<code>true</code>	Indicates that the resulting XDM instance must be typed; that is, the element and attribute nodes must carry the type annotations that result from validation against the schema given at B.1 Schema for the XML Representation of JSON , or against an implementation-defined schema if the <code>liberal</code> option has the value <code>yes</code> .	
<code>false</code>	Indicates that the XDM instance must be untyped.	
<code>escape</code>	Determines whether special characters are represented in the XDM output in backslash-escaped form. The required type is <code>xs:boolean</code> .	

Key	Value	Meaning
	false (default)	All characters in the input that are valid in the version of XML supported by the implementation, whether or not they are represented in the input by means of an escape sequence, are represented as unescaped characters in the result. Any characters or codepoints that are not valid XML characters (for example, unpaired surrogates) are passed to the fallback function as described below; in the absence of a fallback function, they are replaced by the Unicode REPLACEMENT CHARACTER (xFFFF). The attributes escaped and escaped-key will not be present in the XDM output.
	true	<p>JSON escape sequences are used in the result to represent special characters in the JSON input, as defined below, whether or not they were represented using JSON escape sequences in the input. The characters that are considered "special" for this purpose are:</p> <ul style="list-style-type: none"> • all codepoints in the range x00 to x1F or x7F to x9F; • all codepoints that do not represent characters that are valid in the version of XML supported by the processor, including codepoints representing unpaired surrogates; • the backslash character itself (xC). <p>Such characters are represented using a two-character escape sequence where available (for example, \t), or a six-character escape sequence otherwise (for example \uDEAD). Characters other than these will not be escaped in the result, even if they were escaped in the input. In the result:</p> <ul style="list-style-type: none"> • Any string element whose string value contains a backslash character must have the attribute value escaped="true". • Any element that contains a key attribute whose string value contains a backslash character must have the attribute escaped-key="true". • The values of the escaped and escaped-key attributes are immaterial when there is no backslash present, and it is never necessary to include either attribute when its value is false.
fallback		Provides a function which is called when an invalid character is encountered.

Key	Value	Meaning
	Function with signature <pre>function(xs:string) as xs:string</pre>	When an invalid character is encountered this function is called supplying the escaped form of the character as the argument. The function returns a string which is inserted into the result in place of the invalid character. The function also has the option of raising a dynamic error.

The various structures that can occur in JSON are transformed recursively to XDM values according to the rules given in [22.1 XML Representation of JSON](#).

The function returns a document node, whose only child is the element node representing the outermost construct in the JSON text.

The function is not [deterministic](#)^{FO30}: that is, if the function is called twice with the same arguments, it is [implementation-dependent](#) whether the same node is returned on both occasions.

The base URI of the returned document node is taken from the static base URI of the function call.

Error Conditions

[ERR XTDE3240] It is a [dynamic error](#) if the value of \$input does not conform to the JSON grammar as defined by [\[RFC 7159\]](#), allowing implementation-defined extensions if the `liberal` option is set to yes.

[ERR XTDE3245] It is a [dynamic error](#) if the value of the `validate` option is `true` and the processor is not schema-aware.

[ERR XTDE3250] It is a [dynamic error](#) if the value of \$input contains an escaped representation of a character (or codepoint) that is not a valid character in the version of XML supported by the implementation, unless the `unescape` option is set to false.

[ERR XTDE3260] It is a [dynamic error](#) if the value of \$options includes an entry whose key is `liberal`, `validate`, `unescape`, or `fallback`, and whose value is not a permitted value for that key.

Notes

To read a JSON file, this function can be used in conjunction with the [unparsed-text](#)^{FO30} function.

ECMA-404 differs from RFC 4627 in two respects: it does not allow the input to depart from the JSON grammar, but it does allow the top-level construct in the input to be a string, boolean, number, or null, rather than requiring an object or array.

Many JSON implementations allow commas to be used after the last item in an object or array, although the specification does not permit it. The option `spec="liberal"` is provided to allow such deviations from the specification to be accepted. Some JSON implementations also allow constructors such as `new Date("2000-12-13")` to appear as values: specifying `spec="liberal"` allows such extensions to be accepted, but does not guarantee it. If such extensions are accepted, the resulting value is implementation-defined, and will not necessarily conform to the schema at [B.1 Schema for the XML Representation of JSON](#).

Examples

The expression `json-to-xml('{"x": 1, "y": [3,4,5]}')` returns `<map xmlns="http://www.w3.org/2005/xpath-functions"> <number key="x">1</number> <array key="y"> <number>3</number> <number>4</number> <number>5</number> </array> </map>`.

The expression `json-to-xml('"abcd"', map{'liberal': false()})` returns `<string xmlns="http://www.w3.org/2005/xpath-functions">abcd</string>`.

The expression `json-to-xml('{"x": "\\", "y": "\u0025"}')` returns `<map xmlns="http://www.w3.org/2005/xpath-functions"> <string key="x">\</string> <string key="y">%</string> </map>`.

The expression `json-to-xml('{"x": "\\", "y": "\u0025"}', map{'escape': true()})` returns `<map xmlns="http://www.w3.org/2005/xpath-functions"> <string escaped="true" key="x">\</string> <string key="y">%</string> </map>`. (But see the detailed rules for alternative values of the `escaped` attribute on the second `string` element.)

The following example illustrates use of the `fallback` function to handle characters that are invalid in XML.

```
let
  $jsonstr := unparsed-text('http://example.com/endpoint'),
  $options := map {
    'liberal': true(),
    'fallback': function($char as xs:string) as xs:string {
      let
        $c0chars := map {
          '\u0000': '[NUL]',
          '\u0001': '[SOH]',
          '\u0002': '[STX]',
          ...
          '\u001E': '[RS]',
          '\u001F': '[US]'
        },
        $replacement := $c0chars($char)
      return
        if (exists($replacement))
        then $replacement
        else error(xs:QName('err:invalid-char'),
          'Error: ' || $char || ' is not a C0 control character.')
      }
    }
  return json-to-xml($jsonstr, $options)
```

22.4 fn:xml-to-json

Summary

Converts an XML tree, whose format corresponds to the XML representation of JSON defined in this specification, into a string conforming to the JSON grammar.

Signatures

```
fn:xml-to-json($input as node()?) as xs:string?
```

```
fn:xml-to-json($input as node(),
               $options as map(*) as xs:string?)
```

Properties

This function is [deterministic^{FO30}](#), [context-independent^{FO30}](#), and [focus-independent^{FO30}](#).

Rules

The effect of the one-argument form of this function is the same as calling the two-argument form with an empty map as the value of the `$options` argument.

The first argument `$input` is a node; the subtree rooted at this node will typically be the XML representation of a JSON document as defined in [22.1 XML Representation of JSON](#).

If `$input` is the empty sequence, the function returns the empty sequence.

The `$options` argument can be used to control the way in which the conversion takes place. The [option parameter conventions](#) apply.

The entries that may appear in the `$options` map are as follows:

Key	Value	Meaning
<code>indent</code>	Determines whether additional whitespace should be added to the output to improve readability. The required type is <code>xs:boolean</code> .	
<code>false</code>	The processor must not insert any insignificant whitespace between JSON tokens.	
<code>true</code>	The processor <code>MAY</code> insert whitespace between JSON tokens in order to improve readability. The specification imposes no constraints on how this is done.	

The node supplied as `$input` must be one of the following: [\[ERR FOJS0006\]^{FO31}](#)

1. An element node whose name matches the name of a global element declaration in the schema given in [B.1 Schema for the XML Representation of JSON](#) and whose type annotation matches the type of that element declaration (indicating that the element has been validated against this schema).
2. An element node whose name matches the name of a global element declaration in the schema given in [B.1 Schema for the XML Representation of JSON](#), and whose content after stripping all attributes (at any depth) in namespaces other than `http://www.w3.org/2005/xpath-functions` is such that validation against the schema given in [B.1 Schema for the XML Representation of JSON](#) would succeed.

Note:

The reason attributes in alien namespaces are stripped is to avoid the need for a non-schema-aware processor to take into account the effect of attributes such as `xsi:type` and `xsi:nil` that would affect the outcome of schema validation.

3. An element node E having a `key` attribute and/or an `escaped-key` attribute provided that E would satisfy one of the above conditions if the `key` and/or `escaped-key` attributes were removed.
4. A document node having exactly one element child and no text node children, where the element child satisfies any of the conditions above.

Furthermore, `$input` must satisfy the following constraint (which cannot be conveniently expressed in the schema). Every element M that is a descendant-or-self of `$input` and has local name `map` and namespace URI `http://www.w3.org/2005/xpath-functions` must satisfy the following rule: there must not be two distinct children of M (say C_1 and C_2) such that the normalized key of C_1 is equal to the normalized key of C_2 . The normalized key of an element C is as follows:

- If C has the attribute value `escaped-key="true"`, then the value of the `key` attribute of C , with all JSON escape sequences replaced by the corresponding Unicode characters according to the JSON escaping rules.
- Otherwise (the `escaped-key` attribute of C is absent or set to false), the value of the `key` attribute of C .

Nodes in the input tree are handled by applying the following rules, recursively. In these rules the term "an element named N " means "an element node whose local name is N and whose namespace URI is `http://www.w3.org/2005/xpath-functions`".

1. A document node having a single element node child is processed by processing that child.
2. An element named `null` results in the output `null`.
3. An element `$E` named `boolean` results in the output `true` or `false` depending on the result of `xs:boolean(fn:string($E))`.
4. An element `$E` named `number` results in the output of the string result of `xs:string(xs:double(fn:string($E)))`
5. An element named `string` results in the output of the string value of the element, enclosed in quotation marks, with any special characters in the string escaped as described below.
6. An element named `array` results in the output of the children of the `array` element, each processed by applying these rules recursively: the items in the resulting list are enclosed between square brackets, and separated by commas.
7. An element named `map` results in the output of a sequence of map entries corresponding to the children of the `map` element, enclosed between curly braces and separated by commas. Each entry comprises the value of the `key` attribute of the child element, enclosed in quotation marks and escaped as described below, followed by a colon, followed by the result of processing the child element by applying these rules recursively.
8. Comments, processing instructions, and whitespace text node children of `map` and `array` are ignored.

Strings are escaped as follows:

1. If the attribute `escaped="true"` is present for a string value, or `escaped-key="true"` for a key value, then:
 - a. any valid JSON escape sequence present in the string is copied unchanged to the output;
 - b. any invalid JSON escape sequence results in a dynamic error [\[ERR_FOJS0007\]](#)^{FO31};
 - c. any unescaped occurrence of quotation mark, backspace, form-feed, newline, carriage return, tab, or solidus is replaced by `\", \b, \f, \n, \r, \t` or `\/` respectively;

- d. any other codepoint in the range 1-31 or 127-159 is replaced by an escape in the form \uHHHH where HHHH is the upper-case hexadecimal representation of the codepoint value.
2. Otherwise (that is, in the absence of the attribute `escaped="true"` for a string value, or `escaped-key="true"` for a key value):
- a. any occurrence of backslash is replaced by \\
 - b. any occurrence of quotation mark, backspace, form-feed, newline, carriage return, or tab is replaced by \" , \b, \f, \n, \r, or \t respectively;
 - c. any other codepoint in the range 1-31 or 127-159 is replaced by an escape in the form \uHHHH where HHHH is the upper-case hexadecimal representation of the codepoint value.

Error Conditions

A dynamic error is raised [\[ERR FOJS0005\]](#)^{FO31} if the value of `$options` includes an entry whose key is defined in this specification, and whose value is not a permitted value for that key.

A dynamic error is raised [\[ERR FOJS0006\]](#)^{FO31} if the value of `$input` is not a document or element node or is not valid according to the schema for the XML representation of JSON, or if a `map` element has two children whose normalized key values are the same.

A dynamic error is raised [\[ERR FOJS0007\]](#)^{FO31} if the value of `$input` includes a string labeled with `escaped="true"`, or a key labeled with `escaped-key="true"`, where the content of the string or key contains an invalid JSON escape sequence: specifically, where it contains a backslash (\) that is not followed by one of the characters ", \, /, b, f, n, r, t, or u, or where it contains the characters \u not followed by four hexadecimal digits (that is [0-9A-Fa-f]{4}).

Notes

The rule requiring schema validity has a number of consequences, including the following:

1. The input cannot contain no-namespace attributes, or attributes in the namespace <http://www.w3.org/2005/xpath-functions>, except where explicitly allowed by the schema. Attributes in other namespaces, however, are ignored.
2. Nodes that do not affect schema validity, such as comments, processing instructions, namespace nodes, and whitespace text node children of `map` and `array`, are ignored.
3. Numeric values are restricted to those that are valid in JSON: the schema disallows positive and negative infinity and NaN.
4. Duplicate keys within a map are disallowed. Most cases of duplicate keys are prevented by the rules in the schema; additional cases (where the keys are equal only after expanding JSON escape sequences) are prevented by the prose rules of this function. For example, the key values \n and \u000A are treated as duplicates even though the rules in the schema do not treat them as such.

The rule allowing the top-level element to have a `key` attribute (which is ignored) allows any element in the output of the `fn:json-to-xml` function to be processed: for example, it is possible to take a JSON document, convert it to XML, select a subtree based on the value of a `key` attribute, and then convert this subtree back to JSON, perhaps after a transformation. The rule means that an element with the appropriate name will be accepted if it has been validated against one of the types `mapWithinMapType`, `arrayWithinMapType`, `stringWithinMapType`, `numberWithinMapType`, `booleanWithinMapType`, or `nullWithinMapType`.

Examples

The input <array xmlns="http://www.w3.org/2005/xpath-functions"><number>1</number><string>is</string><boolean>1</boolean></array> produces the result [1, "is", true].

The input <map xmlns="http://www.w3.org/2005/xpath-functions"><number key="Sunday">1</number><number key="Monday">2</number></map> produces the result {"Sunday":1, "Monday":2}.

22.5 Transforming XML to JSON

Given an XML structure that does not use the XML representation of JSON defined in [22.1 XML Representation of JSON](#), there are two practical ways to convert it to JSON: either perform a transformation to the XML representation of JSON and then call the [xml-to-json](#) function; or transform it to JSON directly by using custom template rules.

To assist with the second approach, a stylesheet is provided in [B.2 Stylesheet for converting XML to JSON](#). This stylesheet includes a function `j:xml-to-json` which, apart from being in a different namespace, is functionally very similar to the [xml-to-json](#) function described in the previous section. (It differs in doing less validation of the input than the function specification requires, and in the details of how special characters are escaped.) The implementation of the function is exposed, using template rules to perform a recursive descent of the supplied input, and the behavior of the function can therefore be customized (typically by importing the stylesheet and adding additional template rules) to handle arbitrary XML input.

The stylesheet is provided under the W3C software license for the convenience of users. There is no requirement for any conformant XSLT processor to make this stylesheet available. Processors MAY implement the [xml-to-json](#) function by invoking this stylesheet (adapted to achieve full conformance), but there is no requirement to do so.

23 Diagnostics

23.1 Messages

```
<!-- Category: instruction -->
<xsl:message
  select? = expression
  terminate? = { boolean }
  error-code? = { eqname } >
<!-- 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 signal a [dynamic error](#) after sending the message. This error may be caught in the same way as any other dynamic error using [`xsl:catch`](#). 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.

The optional `error-code` attribute (also interpreted as an [attribute value template](#)) may be used to indicate the error code associated with the message. This may be used irrespective of the value of `terminate`. The [effective value](#) of the `error-code` attribute is expected to be an [EQName](#). If no error code is specified, or if the effective value is not a valid EQName, the error code will have local part `XTMM9000` and namespace URI <http://www.w3.org/2005/xqt-errors>. User-defined error codes SHOULD be in a namespace other than <http://www.w3.org/2005/xqt-errors>. When the value of `terminate` is `yes`, the error code may be matched in an [`xsl:catch`](#) element to catch the error and cause processing to continue normally.

[ERR XTMM9000] When a transformation is terminated by use of `<xsl:message terminate="yes"/>`, the effect is the same as when a [dynamic error](#) occurs during the transformation. The default error code is `XTMM9000`; this may be overridden using the `error-code` attribute of the [`xsl:message`](#) instruction.

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>
```

Any [dynamic error](#) that occurs while evaluating the `select` expression or the contained [sequence constructor](#), and any [serialization error](#) that occurs while processing the result, does not cause the transformation to fail; at worst, it means that no message is output, or that the only message that is output is one that relates to the error that occurred.

Note:

An example of such an error is the serialization error that occurs when processing the instruction `<xsl:message select="@code"/>` (on the grounds that free-standing attributes cannot be serialized). Making such errors recoverable means that it is implementation-defined whether or not they are signaled to the user and whether they cause termination of the transformation. If the processor chooses to recover from the error, the content of any resulting message is implementation-dependent.

One possible recovery action is to include a description of the error in the generated message text.

23.2 Assertions

The [`xsl:assert`](#) instruction is used to assert that the value of a particular expression is true; if the value of the expression is false, and assertions are enabled, then a dynamic error occurs.

```
<!-- Category: instruction -->
<xsl:assert
  test = expression
  select? = expression
  error-code? = { eqname } >
  <!-- Content: sequence-constructor -->
</xsl:assert>
```

By default, assertions are disabled.

An implementation MUST provide an external mechanism to enable or disable assertion checking. This may work either statically or dynamically, and may be at the level of the stylesheet as a whole, or at the level of an individual package, or otherwise. The detail of such mechanisms is [implementation-defined](#).

If assertion checking is enabled, the instruction is evaluated as follows:

1. The expression in the `test` attribute is evaluated. If the effective boolean value of the result is `true`, the assertion succeeds, and no further action is taken. If the effective boolean value is `false`, or if a dynamic error occurs during evaluation of the expression, then the assertion fails.
2. If the assertion fails, then the effect of the instruction is governed by the rules for evaluation of an [`xsl:message`](#) instruction with the same `select` attribute, `error-code` attribute, and contained [`sequence-constructors`](#), and with the value `terminate="yes"`. However, the default error code if the `error-code` attribute is omitted is `XTMM9001` rather than `XTMM9000`.

Note:

To the extent that the behavior of [`xsl:message`](#) is [implementation-defined](#), this rule does not prevent an implementation treating [`xsl:assert`](#) and [`xsl:message`](#) differently.

Note:

If evaluation of the `test` expression fails with a dynamic error, the effect is exactly the same as if the evaluation returns `false`, including the fact that the instruction fails with error code `XTMM9001`.

3. If an assertion fails, then the following sibling instructions of the [`xsl:assert`](#) instruction are not evaluated.

Note:

This means that [`xsl:assert`](#) can be used (rather like [`xsl:if`](#) and [`xsl:choose`](#)) to prevent subsequent instructions from executing if a particular precondition is not true, which might be useful if the subsequent instructions have side-effects (for example, by calling extension functions) or if they can fail in uncatchable ways (for example, non-terminating recursion). It is worth noting that there are limits to this guarantee. It does not ensure, for example, that when an assertion within a template fails, the following siblings of the [`xsl:call-template`](#) instruction that invokes that template will not be evaluated; nor does it ensure that if an assertion fails while processing the first item of a sequence using [`xsl:for-each`](#), then subsequent items in the sequence will not be processed.

[ERR XTMM9001] When a transformation is terminated by use of [`xsl:assert`](#), the effect is the same as when a [dynamic error](#) occurs during the transformation. The default error code is `XTMM9001`; this may be overridden using the `error-code` attribute of the [`xsl:assert`](#) instruction.

As with any other dynamic error, an error caused by an assertion failing may be trapped using [xsl:try](#): see [8.3 Try/Catch](#).

The result of the [xsl:assert](#) instruction is an empty sequence.

Example: Using Assertions with Static Parameters

The following example shows a stylesheet function that checks that the value of its supplied argument is in range. The check is performed only if the [static parameter](#) \$DEBUG is set to true.

```
<xsl:param name="DEBUG" as="xs:boolean" select="false()"  
          static="yes" required="no"/>  
<xsl:function name="f:days-elapsed" as="xs:integer">  
  <xsl:param name="date" as="xs:date"/>  
  <xsl:assert use-when="$DEBUG" test="$date lt current-date()"/>  
  <xsl:sequence select="(current-date() - $since)  
                        div xs:dayTimeDuration('PT1D'))"/>  
</xsl:function>
```

Note:

Implementations should avoid optimizing [xsl:assert](#) instructions away. As a guideline, if the result of a sequence constructor is required by the transformation, the implementation should ensure that all [xsl:assert](#) instructions in that sequence constructor are evaluated. Conversely, if the result of a sequence constructor is not required by the transformation, its [xsl:assert](#) instructions should not be evaluated.

This guidance is not intended to prevent optimizations such as lazy evaluation, where evaluation of a sequence constructor may finish early, as soon as enough information is available to evaluate the containing instruction.

An implementation MAY provide a user option allowing a processor to treat assertions as being true without explicit checking. This option MUST NOT be enabled by default. If such an option is in force, the effect of any assertion not being true is [implementation-dependent](#).

Note:

For example, given the assertion `<xsl:assert test="count(//title)=1"/>`, a processor might generate code for the expression `<xsl:value-of select="//title"/>` that stops searching for `title` elements after finding the first one. In the event that the source document contains more than one `title`, execution of the stylesheet may fail in arbitrary ways, or it may produce incorrect output.

24 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 named function introduced to the static or dynamic context by mechanisms outside the scope of this specification.]

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.

[ERR XTSE0085] It is a [static error](#) to use a [reserved namespace](#) in the name of any [extension function](#) or [extension instruction](#), other than a function or instruction defined in this specification or in a normatively referenced specification. It is a [static error](#) to use a prefix bound to a reserved namespace in the `[xsl:]extension-element-prefixes` attribute.

24.1 Extension Functions

The set of functions that can be called from a [FunctionCall](#)^{XP30} 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, which MUST NOT be the URI of a [reserved namespace](#).

Note:

The definition of the term [extension function](#) is written to exclude user-written [stylesheet functions](#), constructor functions for built-in and user-defined types, functions in the `fn`, `math`, `map`, and `array` namespaces, anonymous XPath inline functions, [maps](#) and arrays (see [27.7.1 Arrays](#)), and partial function applications (including partial applications of extension functions). It also excludes functions obtained by invoking XPath-defined functions such as [load-xquery-module](#)^{FO31}. The definition allows extension functions to be discovered at evaluation time (typically using [function-lookup](#)^{FO30}) rather than necessarily being known statically.

Technically, the definition of extension functions excludes anonymous functions obtained by calling or partially applying other extension functions. Since such functions are by their nature implementation-defined, they may however share some of the characteristics of extension functions.

24.1.1 [fn:function-available](#)

Summary

Determines whether a particular function is or is not available for use. The function is particularly useful for calling within an `[xsl:]use-when` attribute (see [3.13.1 Conditional Element Inclusion](#)) to test whether a particular [extension function](#) is available.

Signatures

```
fn:function-available($function-name as xs:string) as xs:boolean
```

```
fn:function-available($function-name as xs:string,
                     $arity           as xs:integer) as xs:boolean
```

Properties

This function is [deterministic^{FO30}](#), [context-dependent^{FO30}](#), and [focus-independent^{FO30}](#). It depends on namespaces, and known function signatures.

Rules

A function is said to be available within an XPath expression if it is present in the [statically known function signatures^{XP30}](#) for that expression (see [5.3.1 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 an [E QName](#). The lexical QName is expanded into an [expanded QName](#) using the namespace declarations in scope for the [expression](#). If the value is an unprefixed lexical QName, 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.

When the containing expression is evaluated with [XPath 1.0 compatibility mode](#) set to true, 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>
```

Error Conditions

[ERR_XTDE1400] It is a [dynamic error](#) if the argument does not evaluate to a string that is a valid [E QName](#), or if the value is a [lexical QName](#) with a prefix for which no namespace declaration is present in the static context. 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](#).

Notes

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.

The introduction of the `function-lookupFO30` function in XPath 3.0 reduces the need for `function-available`, since `function-lookupFO30` not only tests whether a function is available, but also returns a function item that enables it to be dynamically called.

If a function is present in the static context but with no useful functionality (for example, if the system has been configured for security reasons so that `available-environment-variablesFO30` returns no information), then `function-available` when applied to that function should return false.

It is not necessary that there be a direct equivalence between the results of `function-available` and `function-lookupFO30` in all cases. For example, there may be `extension functions` whose side-effects are such that for security reasons, dynamic calls to the function are disallowed; `function-lookupFO30` might then not provide access to the function. The main use-case for `function-available`, by contrast, is for use in [xsl:]use-when conditions to test whether static calls on the function are possible.

Examples

Example: Stylesheet portable between XSLT 1.0, XSLT 2.0, and XSLT 3.0

A stylesheet that is designed to use XSLT 2.0 facilities when running under an XSLT 2.0 or XSLT 3.0 processor, 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(/doc/title, '[a-z]*')"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="string-length(
        translate(/doc/title, 'abcdefghijklmnopqrstuvwxyz', ''))"
      = 0"/>
    </xsl:otherwise>
  </xsl:choose>
</out>
```

Here an XSLT 2.0 or XSLT 3.0 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 3.0 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 or XSLT 3.0 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(/doc/title, 10)"
  use-when="function-available('pad', 2)"/>
<xsl:value-of select="concat(/doc/title, string-join(
  for $i in 1 to 10 - string-length(/doc/title)
  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.

24.1.2 Calling Extension Functions

If the function name used in a [FunctionCall^{XP30}](#) within an XPath [expression](#) identifies an extension function, then to evaluate the [FunctionCall^{XP30}](#), the processor will first evaluate each of the arguments in the [FunctionCall^{XP30}](#). If the processor has information about the datatypes expected by the extension function, then it **MAY** perform any necessary type conversions between the XPath datatypes 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 datatypes 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 [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 the containing element is processed with [XSLT 1.0 behavior](#), it is a [dynamic error](#) to evaluate an extension function call if no implementation of the extension function is available.

Note:

When XSLT 1.0 behavior is not enabled, this is a static error [\[ERR XPST0017\]^{XP30}](#).

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](#).

24.1.3 [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, whether as an atomic value, a node, or a function item. 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 datatype, or they might be mapped to existing datatypes such as `integer`, `string`, or `anyURI`.

24.1.4 [fn:type-available](#)

Summary

Used to control how a stylesheet behaves if a particular schema type is or is not available in the static context.

Signature

```
fn:type-available($type-name as xs:string) as xs:boolean
```

Properties

This function is [deterministic](#)^{FO30}, [context-dependent](#)^{FO30}, and [focus-independent](#)^{FO30}. It depends on namespaces, and schema definitions.

Rules

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](#)^{XP30} for that expression (see [5.3.1 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 an [EQName](#). The EQName is expanded into an [expanded QName](#) using the namespace declarations in scope for the [expression](#). If the value is an unprefixed lexical QName, 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.

Error Conditions

[ERR XTDE1428] It is a [dynamic error](#) if the argument does not evaluate to a string that is a valid [EQName](#), or if the value is a [lexical QName](#) with a prefix for which no namespace declaration is present in the static context. 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](#).

Notes

The [type-available](#) function is of limited use within an [xsl:]use-when expression, because the static context for the expression does not include any user-defined types.

24.2 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 [3.7.3 User-defined Data Elements](#)) are not extension elements as defined here, and nothing in this section applies to them.

24.2.1 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 [3.4 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 `xmllns`) may be designated as an extension namespace by including `#default` in the list of namespace prefixes.

A [reserved namespace](#) cannot be designated as an extension namespace: see [see [ERR XTSE0085](#)].

[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.

24.2.2 [fn:element-available](#)

Summary

Determines whether a particular instruction is or is not available for use. The function is particularly useful for calling within an [xsl:]use-when attribute (see [3.13.1 Conditional Element Inclusion](#)) to test whether a particular [extension instruction](#) is available.

Signature

```
fn:element-available($element-name as xs:string) as xs:boolean
```

Properties

This function is [deterministic](#)^{FO30}, [context-dependent](#)^{FO30}, and [focus-independent](#)^{FO30}. It depends on namespaces.

Rules

The value of the \$element-name argument MUST be a string containing an [EQName](#). If it is a [lexical QName](#) with a prefix, then it is expanded into an [expanded QName](#) using the namespace declarations in the static context of the [expression](#). If there is a default namespace in scope, then it is used to expand an unprefixed [lexical QName](#).

If the resulting [expanded QName](#) is in the [XSLT namespace](#), the function returns true if and only if the local name matches the name of an XSLT element that is defined in this specification and implemented by the XSLT processor.

If the [expanded QName](#) has a null namespace URI, the [element-available](#) function will return false.

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 [24.2.3 Fallback](#). An implementation MUST NOT signal an error merely because the stylesheet contains an extension instruction for which no implementation is available.

Error Conditions

[ERR XTDE1440] It is a [dynamic error](#) if the argument does not evaluate to a string that is a valid [EQName](#), or if the value is a [lexical QName](#) with a prefix for which no namespace declaration is present in the static context.

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](#).

Notes

For element names in the XSLT namespace:

- This function can be useful to distinguish processors that implement XSLT 3.0 from processors that implement other (older or newer) versions of the specification, and to distinguish full implementations from incomplete implementations. (Incomplete implementations, of course, cannot be assumed to behave as described in this specification.)
- In earlier versions of this specification, [element-available](#) was defined to return true only for elements classified as instructions. The distinction between instructions and other elements, however, is sometimes rather technical, and in XSLT 3.0 the effect of the function has therefore been aligned to do what its name might suggest.
- If an instruction is recognized but offers no useful functionality (for example, if the system has been configured for security reasons so that [xsl:evaluate](#) always raises an error), then [element-available](#) when applied to that instruction SHOULD return false.

For element names in other namespaces:

- The result of the [element-available](#) does not depend on whether or not the namespace of the supplied instruction name has been designated as an extension element namespace; it tests whether the instruction would be available if the namespace were designated as such.

24.2.3 [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 not evaluated.

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 3.0, 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 [dynamic error](#) if it has no [xsl:fallback](#) children.

Note:

This is different from the situation with unrecognized [XSLT elements](#). As explained in [3.10 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.

25 Transformation Results

The output of a transformation includes a [principal result](#) and zero or more [secondary results](#).

The way in which these results are delivered to an application is [implementation-defined](#).

Serialization of results is described further in [26 Serialization](#)

25.1 Creating Secondary Results

```
<!-- Category: instruction -->
<xsl:result-document
    format? = { eqname }
    href? = { uri }
    validation? = "strict" | "lax" | "preserve" | "strip"
    type? = eqname
    method? = { "xml" | "html" | "xhtml" | "text" | "json" | "adaptive" | eqname }
    allow-duplicate-names? = { boolean }
    build-tree? = { boolean }
    byte-order-mark? = { boolean }
    cdata-section-elements? = { eqnames }
    doctype-public? = { string }
    doctype-system? = { string }
    encoding? = { string }
    escape-uri-attributes? = { boolean }
    html-version? = { decimal }
    include-content-type? = { boolean }
    indent? = { boolean }
    item-separator? = { string }
    json-node-output-method? = { "xml" | "html" | "xhtml" | "text" | eqname }
    media-type? = { string }
    normalization-form? = { "NFC" | "NFD" | "NFKC" | "NFKD" | "fully-normalized" |
    "none" | nmtoken }
    omit-xml-declaration? = { boolean }
    parameter-document? = { uri }
    standalone? = { boolean | "omit" }
    suppress-indentation? = { eqnames }
    undeclare-prefixes? = { boolean }
    use-character-maps? = eqnames
    output-version? = { nmtoken } >
    <!-- Content: sequence-constructor -->
</xsl:result-document>
```

The `xsl:result-document` instruction is used to create a [secondary result](#). The content of the `xsl:result-document` element is a [sequence constructor](#), and the value of the [secondary result](#) (known as the [raw result](#)) is the [immediate result](#) of this sequence constructor.

As with the [principal result](#) of the transformation, a [secondary result](#) may be delivered to the calling application in three ways (see [2.3.6 Post-processing the Raw Result](#)):

1. The [raw result](#) may be delivered *as is*.
2. The [raw result](#) may be used to construct a [final result tree](#) by invoking the process of [sequence normalization](#)^{SER30}.
3. The [raw result](#) may be serialized to a sequence of octets (which may then, optionally, be saved to a persistent storage location).

The decision whether or not to serialize the raw result depends on the [processor](#) and on the way it is invoked. This is [implementation-defined](#), and it is not controlled by anything in the stylesheet.

If the result is not serialized, then the decision whether to return the [raw result](#) or to construct a tree depends on the effective value of the `build-tree` attribute. If the effective value of the `build-tree` attribute is yes, then a [final result tree](#) is created by invoking the process of [sequence normalization](#)^{SER30}. The default for the `build-tree` attribute depends on the serialization method. For the `xml`, `html`, `xhtml`, and `text` methods the default value is yes. For the `json` and `adaptive` methods (available only with XPath 3.1) the default value is no.

The [`xsl:result-document`](#) instruction defines a URI that may be used to identify the [secondary result](#). The instruction may optionally specify the output format to be used for serializing the result.

Technically, the result of evaluating the [`xsl:result-document`](#) instruction is an empty sequence. This means it does not contribute anything to the result of the sequence constructor it is part of.

The [effective value](#) of the `format` attribute, if specified, MUST be an [EQName](#). The value is expanded using the namespace declarations in scope for the [`xsl:result-document`](#) element. The resulting [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 [26 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 [dynamic error](#) if the [effective value](#) of the `format` attribute is not a valid [EQName](#), or if it does not match the [expanded QName](#) of an [output definition](#) in the containing [package](#). 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 `parameter-document` attribute allows serialization parameters to be supplied in an external document. The external document must contain an `output:serialization-parameters` element with the format described in [Section 3.1 Setting Serialization Parameters by Means of a Data Model Instance](#)^{SER30}, and the parameters are interpreted as described in that specification.

If present, the [effective value](#) of the URI supplied in the `parameter-document` attribute is dereferenced, after resolution against the base URI of the [`xsl:result-document`](#) element if it is a relative reference. The parameter document SHOULD be read during run-time evaluation of the stylesheet. If the location of the stylesheet at development time is different from the deployed location, any relative reference should be resolved against the deployed location. A serialization error occurs if the result of dereferencing the URI is ill-formed or invalid; but if no document can be found at the specified location, the attribute should be ignored.

A serialization parameter specified in the `parameter-document` takes precedence over a value supplied directly as an attribute of [`xsl:result-document`](#), which in turn takes precedence over a value supplied in the selected output definition, except that the values of the `cdata-section-elements` and `suppress-indentation` attributes are merged in the same way as when multiple [`xsl:output`](#) declarations are merged.

The attributes `method`, `allow-duplicate-names`, `build-tree`, `byte-order-mark` `cdata-section-elements`, `doctype-public`, `doctype-system`, `encoding`, `escape-uri-attributes`, `html-version`,

`indent`, `item-separator`, `json-node-output-method`, `media-type`, `normalization-form`, `omit-xml-declaration`, `standalone`, `suppress-indentation`, `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. Some of the attributes have more specific rules:

- In the case of `cdata-section-elements` and `suppress-indentation`, 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 the case of `doctype-public` and `doctype-system`, setting the effective value of the attribute to a zero-length string has the effect of overriding any value for these attributes obtained from the output definition. The corresponding serialization parameter is not set (is “absent”).
- In the case of `item-separator`, setting the effective value of the attribute to the special value “#absent” has the effect of overriding any value for this attribute obtained from the output definition. The corresponding serialization parameter is not set (is “absent”). It is not possible to set the value of the serialization parameter to the literal 7-character string “#absent”.
- 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`, `suppress-indentation`, and `use-character-maps`, the [effective value](#) of the attribute contains a space-separated list of [EQNames](#). If any of these is a [lexical QName](#) with a prefix, the prefix is expanded using the in-scope namespaces for the [`xsl:result-document`](#) element. In the case of `cdata-section-elements` and `suppress-indentation`, an unprefixed element name is expanded using the default namespace. In the case of the `method` attribute, if the method is not one of the system-defined methods (`xml`, `html`, `xhtml`, `text`) then the expanded name must have a non-absent namespace.

Unless the processor implements the [XPath 3.1 Feature](#), the `method` values `json` and `adaptive` MUST be rejected as invalid, and the attributes `allow-duplicate-names` and `json-node-output-method` MUST be ignored. The meaning of these output methods and serialization parameters is defined in [\[XSLT and XQuery Serialization 3.1\]](#).

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 [3.4 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 `item-separator` serialization parameter is used when the `raw result` is used to construct a result tree by applying sequence normalization, and it is also used when the result tree is serialized. For example, if the sequence constructor delivers a sequence of integers, and the `text` serialization method is used, then the result of serialization will be a string obtained by converting each integer to a string, and separating the strings using the defined `item-separator`.

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. If it is relative, then it is resolved against the base output URI. 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 valid relative URI reference **MUST** be accepted. Note that the zero-length string is a valid relative URI reference.

If the implementation provides an API to access secondary results, then it **MUST** allow a secondary result to be identified by means of the absolutized value of the `href` attribute. In addition, if a final result tree is constructed (that is, if the effective value of `build-tree` is `yes`), then this value is used as the base URI of the document node at the root of the final result tree.

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 reference. 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 a final result tree, 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 [25.4.2 Validating Document Nodes](#). Any such validation is applied to the document node produced as the result of sequence normalization^{SER30}. If sequence normalization does not take place (typically because the `raw result` is delivered to the application directly, or because the selected serialization method does not involve sequence normalization) then the `validation` and `type` attributes are ignored.

Note:

Validation applies after inserting item separators as determined by the `item-separator` serialization parameter, and an inappropriate choice of `item-separator` may cause the result to become invalid.

A [processor](#) MAY allow a [final result tree](#) to be serialized. Serialization is described in [26 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="3.0"
    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>

```

25.2 Restrictions on the use of `xsl:result-document`

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 named 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:function](#), [xsl:key](#), [xsl:sort](#), [xsl:accumulator-rule](#), and [xsl:merge-key](#) always evaluate the instructions in their contained [sequence constructor](#) in [temporary output state](#).

[ERR XTDE1480] It is a [dynamic error](#) to evaluate the [xsl:result-document](#) instruction in [temporary output state](#).

[ERR XTDE1490] It is a [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.

In addition, an implementation MAY report this error if it is able to detect that two or more final result trees are generated with different URIs that refer to the same physical resource.

[ERR XTDE1500] It is a [dynamic error](#) for a [stylesheet](#) to write to an external resource and read from the same resource during a single transformation, if the same absolute URI is used to access the resource in both cases.

In addition, an implementation MAY report this error if it is able to detect that a transformation writes to a resource and reads from the same resource using different URIs that refer to the same physical resource. Note that if the error is not detected, it is [implementation-dependent](#) whether the document that is read from the resource reflects its state before or after the result tree is written.

25.3 [The Current Output URI](#)

[**DEFINITION:** The **current output URI** is the URI associated with the [principal result](#) or [secondary result](#) that is currently being written.]

25.3.1 [fn:current-output-uri](#)

Summary

Returns the value of the [current output URI](#).

Signature

```
fn:current-output-uri() as xs:anyURI?
```

Properties

This function is [deterministic](#)^{FO30}, [focus-independent](#)^{FO30}, and [context-dependent](#)^{FO30}.

Rules

On initial invocation of a stylesheet component, the current output uri is set to the [base output URI](#).

During execution of an [xsl:result-document](#) instruction with an href attribute, the current output URI changes to the absolute URI obtained by resolving the [effective value](#) of the href attribute against the base output URI.

The current output URI is cleared (set to [absent](#)) while evaluating stylesheet functions, dynamic function calls, evaluation of global variables, stylesheet parameters, and patterns. If the function is called when the current output URI is absent, the function returns the empty sequence.

The current output URI may also be [absent](#) in the event that a stylesheet is invoked without supplying a [base output URI](#).

Notes

The current output URI is not cleared when evaluating a local variable, even though [xsl:result-document](#) cannot be used while evaluating a local variable. The reason for this is to allow the value of [current-output-uri](#) to be set as the value of a tunnel parameter, so that the original base output URI is accessible even when writing nested result documents.

25.4 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 same attributes are used on [xsl:source-document](#) and [xsl:merge-source](#) to control validation of input documents.

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 [xsl:]default-validation attribute of the innermost containing element having such an attribute; if this is not specified, the effect is the same as specifying validation="strip".

The [xsl:]default-validation attribute defines the default value of the validation attribute of all [xsl:element](#), [xsl:attribute](#), [xsl:copy](#), [xsl:copy-of](#), [xsl:document](#), and [xsl:result-document](#) instructions, and of the [xsl:validation](#) attribute of all [literal result elements](#), appearing within its scope. 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 containing [stylesheet module](#) or [package](#): it does not extend to included or imported stylesheet modules or used packages. If the attribute is omitted, the default is `strip`. The permitted values are `preserve` and `strip`.

The `[xsl:]default-validation` attribute has no effect on the [`xsl:source-document`](#) and [`xsl:merge-source`](#) elements, which perform no validation unless explicitly requested.

[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`](#), [`xsl:result-document`](#), [`xsl:source-document`](#), or [`xsl:merge-source`](#) elements, 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 [25.4.1 Validating Constructed Elements and Attributes](#), while those for document nodes are given in [25.4.2 Validating Document Nodes](#).

25.4.1 [Validating Constructed Elements and Attributes](#)

25.4.1.1 [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.

1. 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.
2. 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:
 - a. 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. The `nilled`, `is-id` and `is-idrefs` properties on the new element are set to `false`.
 - b. 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`.

The `is-id` and `is-idrefs` properties on the new attribute are set to `false`.

- c. In the case of `xsl:copy-of`, all the nodes that are copied will retain their type annotations unchanged. The values of their `nilled`, `is-id` and `is-idrefs` properties are also unchanged.
 - d. In the case of `xsl:copy`, the effect depends on the kind of node being copied.
 - i. Where the node being copied is an attribute, the copied attribute will retain its `type annotation` and the values of its `is-id` and `is-idrefs` properties.
 - ii. 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 values of the `nilled`, `is-id`, and `is-idrefs` properties are handled in the same way as `xsl:element`.
3. 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:

- a. 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). 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.

- b. 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.
- c. 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 [3.15 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.

- d. The type annotations on the resulting nodes, as well as the values of their `is-id`, `is-idrefs`, and `nilled` properties, are defined by the rules in [Section 3.3 Construction from a PSVI](#)^{DM31}.
- e. 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.
- 4. 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.

The type annotations on the resulting nodes, as well as the values of their `is-id`, `is-idrefs`, and `nilled` properties, are defined by the rules in [Section 3.3 Construction from a PSVI](#)^{DM31}.

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 [25.4.1.2 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.

25.4.1.2 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 datatype).
- If the element or attribute is not considered valid, as defined above, the transformation fails [see [ERR_XTTE1540](#)].

If an element node is validated against the type **xs:untyped**, the effect is the same as specifying **validation="strip"**: that is, the elements and attributes in the subtree rooted at the target element are copied with a type annotation of **xs:untyped** or **xs:untypedAtomic** respectively.

If an element or attribute node is validated against the type **xs:untypedAtomic**, the effect is the same as specifying **[xsl:]type="xs:string"** except that when validation succeeds, the returned element or attribute has a type annotation of **xs:untypedAtomic**. Validation fails in the case of an element with element children.

[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 [package](#).

[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_XTTE1535] It is a [type error](#) if the value of the **type** attribute of an [xsl:copy](#) or [xsl:copy-of](#) instruction refers to a complex type definition and one or more of the items being copied is an attribute node.

[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.

25.4.1.3 [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" SHOULD be 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 3.0 provides no facility to construct unparsed entities within a 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 [\[XDM 3.0\]](#) ([Section 3.3.1 Mapping PSVI Additions to Node Properties](#)^{DM30}). This process creates nodes with simple or complex [type annotations](#) based on the types established during schema validation.

The above process must be done in such a way that the base URI property of every node in the resulting XDM tree is the same as the base URI property of the corresponding node in the input tree.

Note:

As an alternative to steps 1 and 2, the XDM tree may be converted to an Infoset directly, using the mapping rules given for each kind of node in [\[XDM 3.0\]](#) (Section 6).

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`.

[25.4.2 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 XSLT elements [xsl:source-document](#), [xsl:merge-source](#), [xsl:document](#), [xsl:result-document](#), [xsl:copy](#), or [xsl:copy-of](#), and this element 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](#) or [xsl:assert](#).

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 [25.4.1 Validating Constructed Elements and Attributes](#).

Note:

The `type` attribute on `xsl:source-document`, `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 3.0 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.
- 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 (such as ID/IDREF constraints) are not satisfied.

25.4.3 [Validating `xml:id` attributes](#)

This section provides a non-normative summary of the effect of validation on attributes named `xml:id`. The normative rules can be inferred from rules given elsewhere in this section.

1. When an attribute named `xml:id` is encountered in the course of validation:
 - A validation error occurs if it the attribute is not lexically valid against type `xs:ID`.
 - The typed value of the attribute is whitespace-normalized.
 - The attribute is labeled with type annotation `xs:ID`.
 - The attribute acquires the `is-id` property.
2. The previous rule applies whether validation is strict, lax, or by type; validation will never fail (or be skipped) on the grounds that no global attribute declaration named `xsl:id` is available.
3. Checking `xml:id` attributes for uniqueness happens if and only if validation is applied at the level of a document node.

26 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 [27 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? = eqname
    method? = "xml" | "html" | "xhtml" | "text" | "json" | "adaptive" | eqname
    allow-duplicate-names? = boolean
    build-tree? = boolean
    byte-order-mark? = boolean
    cdata-section-elements? = eqnames
    doctype-public? = string
    doctype-system? = string
    encoding? = string
    escape-uri-attributes? = boolean
    html-version? = decimal
    include-content-type? = boolean
    indent? = boolean
    item-separator? = string
    json-node-output-method? = "xml" | "html" | "xhtml" | "text" | eqname
    media-type? = string
    normalization-form? = "NFC" | "NFD" | "NFKC" | "NFKD" | "fully-normalized" |
    "none" | nmtoken
    omit-xml-declaration? = boolean
    parameter-document? = uri
    standalone? = boolean | "omit"
    suppress-indentation? = eqnames
    undeclare-prefixes? = boolean
    use-character-maps? = eqnames
    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 within a [package](#) 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.]

An output definition is scoped to a package. If this is a [library package](#) the output definition applies only to [xsl:result-document](#) instructions within the same package. If it is the [top-level package](#), the output definition applies to [xsl:result-document](#) instructions within the same package and also to the implicit [final result tree](#).

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 [principal result](#).

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` and `suppress-indentation` attributes, 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](#).

The `parameter-document` attribute allows serialization parameters to be supplied in an external document. The external document must contain an `output:serialization-parameters` element with the format described in [Section 3.1 Setting Serialization Parameters by Means of a Data Model Instance](#)^{SER30}, and the parameters are interpreted as described in that specification.

If present, the URI supplied in the `parameter-document` attribute is dereferenced, after resolution against the base URI of the [xsl:output](#) element if it is a relative reference. The parameter document `SHOULD` be read during static analysis of the stylesheet. A serialization error occurs if the result of dereferencing the URI is ill-formed or invalid; but if no document can be found at the specified location, the attribute `SHOULD` be ignored.

A serialization parameter specified in the `parameter-document` takes precedence over a value supplied directly in the output declaration, except that the values of the `cdata-section-elements` and `suppress-indentation` attributes are merged in the same way as when multiple [xsl:output](#) declarations are merged.

[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`, `suppress-indentation`, 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.

The `build-tree` attribute controls whether the raw [principal result](#) or [secondary result](#) is converted to a [final result tree](#). The default depends on the value of the `method` attribute: the default is `yes` if the `method` attribute specifies `xml`, `html`, `xhtml`, or `text`, or if it is omitted; the default is `no` if the `method` attribute specifies `json` or `adaptive`. A [final result tree](#) may be constructed whether or not it is subsequently serialized.

Note:

The default for `build-tree` may differ for user-defined serialization methods or for serialization methods introduced in future versions of this specification.

Unless the processor implements the [XPath 3.1 Feature](#), the `method` values `json` and `adaptive` MUST be rejected as invalid, and the attributes `allow-duplicate-names` and `json-node-output-method` MUST be ignored. The meaning of these output methods and serialization parameters is defined in [\[XSLT and XQuery Serialization 3.1\]](#).

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 URI references in the `href` attribute of the `xsl:result-document` instruction, then these relative URI references 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 [EQName](#). If it is a [lexical QName](#) with no prefix, then it identifies a method specified in [\[XSLT and XQuery Serialization\]](#) and MUST be one of `xml`, `html`, `xhtml`, or `text`. If it is a [lexical QName](#) with a prefix, then the [lexical QName](#) is expanded into an [expanded QName](#) as described in [5.1.1 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 [effective version](#) of the outermost element of the [principal stylesheet module](#) in the [top-level package](#) 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. If the attribute is absent or has a zero-length string as its value, then the serialization parameter is not set (is “absent”).
- The value of the `doctype-public` attribute provides the value of the `doctype-public` parameter to the serialization method. If the attribute is absent or has a zero-length string as its value, then the serialization parameter is not set (is “absent”).

The value of `doctype-public` must conform to the rules for a [PubidLiteral^{XML}](#) (see [\[XML 1.0\]](#)).

- 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 `html-version` attribute provides the value of the `html-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.

Note:

This serialization parameter is new in version 3.0. If it is absent, the `html` output method uses the value of the `version` parameter in its place. For XHTML serialization, the `html-version` parameter indicates the version of XHTML to be used, while the `version` parameter indicates the version of XML.

- 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 `item-separator` attribute provides the value of the `item-separator` parameter to the serialization method. The value of the serialization parameter can be any string (including a zero-length string), or absent. To set the parameter to absent, the `item-separator` attribute can either be omitted, or set to the special value `item-separator="#absent"`; it is not possible to set the value of the serialization parameter to the literal 7-character string `#absent`.

Note:

The `item-separator` attribute has no effect if the sequence being serialized contains only one item, which will always be the case if the effective value of `build-tree` is `yes`.

- 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 specifies 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 `suppress-indentation` 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 `suppress-indentation` 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 `undeclare-prefixes` attribute provides the value of the `undeclare-prefixes` parameter to the serialization method. The default value is `no`.
- 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 [26.1 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 serialization errors that occur. These have the same effect as [dynamic errors](#): that is, the processor must signal the error and must not finish as if the transformation had been successful.

26.1 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 = eqname
  use-character-maps? = eqnames >
  <!-- 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.

The name of a character map is local to the [package](#) in which its declaration appears; it may be referenced only from within the same package.

[ERR XTSE1580] It is a [static error](#) if a [package](#) 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 containing [package](#).

[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 [26.2 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="&nbsp;"/>
  <xsl:output-character character="¡" string="&iexcl;"/>
  ...
</xsl:character-map>

<xsl:character-map name="doc-entities">
  <xsl:output-character character="" string="&t-and-c;"/>
  <xsl:output-character character="" string="&chap1;"/>
  <xsl:output-character character="" string="&chap2;"/>
  ...
</xsl:character-map>

<xsl:character-map name="windows-format">
  <!-- newlines as CRLF -->
  <xsl:output-character character="
" string="&#xD;&#xA;"/>

  <!-- tabs as three spaces -->
  <xsl:output-character character="	" string="    "/>

  <!-- images for special characters -->
  <xsl:output-character character="" string="&lt;img src='special1.gif' /&gt;"/>
  <xsl:output-character character="" string="&lt;img src='special2.gif' /&gt;"/>
  ...
</xsl:character-map>
```

Note:

When character maps are used, there is no guarantee that the serialized output will be well-formed XML (or HTML). Furthermore, the fact that the result tree was validated against a schema gives no guarantee that the serialized output will still be valid against the same schema. Conversely, it is possible to use character maps to produce schema-valid output from a result tree that would fail validation.

26.2 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 an 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 that the serializer (described in [\[XSLT and XQuery Serialization\]](#)) be extended as follows.

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-escaping` 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.

Similarly, 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 request to disable output escaping is ignored.

Note:

Furthermore, a request to disable output escaping has no effect when the newly constructed text node is used to form the value of an attribute, comment, processing instruction, or namespace node. This is because the rules for constructing such nodes (see [5.7.2 Constructing Simple Content](#)) cause the text node to be atomized, and the process of atomizing a text node takes no account of the disable-escaping property.

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 this method 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. If disabling output escaping is not supported, any request to disable output escaping is ignored.

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:

When `disable-output-escaping` is used, there is no guarantee that the serialized output will be well-formed XML (or HTML). Furthermore, the fact that the result tree was validated against a schema gives no guarantee that the serialized output will still be valid against the same schema. Conversely, it is possible to use `disable-output-escaping` to produce schema-valid output from a result tree that would fail validation.

Note:

The facility to define character maps for use during serialization, as described in [26.1 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.

27 Conformance

A [processor](#) that claims conformance with this specification **MUST** satisfy the conformance requirements for a [basic XSLT processor](#) and for each of the optional features with which it claims conformance.

The following optional features are defined:

1. The schema-awareness feature, defined in [27.2 Schema-Awareness Conformance Feature](#)
2. The serialization feature, defined in [27.3 Serialization Feature](#)
3. The backwards compatibility feature, defined in [27.4 Compatibility Features](#)
4. The streaming feature, defined in [27.5 Streaming Feature](#).
5. The dynamic evaluation feature, defined in [27.6 Dynamic Evaluation Feature](#).
6. The higher-order functions feature, defined in [27.8 Higher-Order Functions Feature](#).
7. The XPath 3.1 feature, defined in [27.7 XPath 3.1 Feature](#).

A processor that does not claim conformance with an optional feature **MUST** satisfy the requirements for processors that do not implement that feature.

Note:

There is no conformance level or feature defined in this specification that requires implementation of the static typing features described in [\[XPath 3.0\]](#). 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 3.0 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 zero or more XDM trees conforming to the data model defined in [\[XDM 3.0\]](#). 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 feature, 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 [3.10 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, and is not caught using [xsl:catch](#), the processor **MUST** signal it and **MUST** eventually terminate abnormally.

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.

The mandatory requirements of this specification are taken to include the mandatory requirements of [\[XPath 3.0\]](#), [\[XDM 3.0\]](#), and [\[Functions and Operators 3.0\]](#). An XSLT 3.0 processor **MUST** provide a mode of operation which conforms to the 3.0 versions of those specifications as extended by [21 Maps](#) and [22 Processing JSON Data](#).

A processor **MAY** also provide a mode of operation which conforms to the 3.1 versions of those specifications; in this case it must do so as described in [XPath 3.1 Feature](#).

A processor **MAY** also provide a mode of operation which conforms to versions of those specifications later than the 3.1 versions; in such cases the detail of how XSLT 3.0 interacts with new features introduced by such later versions (for example, extensions to the data model) is [implementation-defined](#).

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.

Some of the optional features are defined in such a way that if the feature is not provided, the data model is constrained to exclude certain kinds of item. For example:

- A processor that does not provide the [schema-awareness](#) feature restricts the data model so that it does not contain atomic values of types other than the built-in types, or nodes with non-trivial type annotations.
- A processor that does not provide the [higher-order functions feature](#) constrains the data model so that it does not contain function items other than maps or arrays.
- A processor that does not provide the [XPath 3.1 Feature](#) constrains the data model so that it does not contain arrays.

[ERR XTDE1665] A [dynamic error](#) **MAY** be raised if the input to the processor includes an item that requires availability of an optional feature that the processor does not provide.

Note:

It is not necessarily possible to trigger this error. A processor that does not provide an optional feature might not define or recognize any representation of the items that are disallowed. The error code is provided for use in cases where a processor is able to interoperate with other software that does not have the same constraints — for example, where a package compiled with a non-schema-aware processor is able to invoke functions in a package that was compiled with a schema-aware processor. Even in that case, processors have the option of filtering or converting the input so that it meets the relevant constraints: for example, a non-schema-aware processor when presented with a schema-validated document in the form of a PSVI might simply ignore the properties it does not understand.

The dynamic error is optional: for example a processor might report no error if the offending item is not actually used.

The phrase *input to the processor* is deliberately wide: it includes (inter alia) the [global context item](#), items present in the [initial match selection](#), items passed as [stylesheet parameters](#), items returned by functions such as [document](#), [doc^{FO30}](#), and [collection^{FO30}](#), items returned by [extension functions](#) and [extension instructions](#), items supplied in function or template parameters or results across package boundaries, and nodes reachable from any of the above by axis navigation.

[27.1 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 constructs explicitly associated with an optional feature.] These constructs are listed below.

27.2 Schema-Awareness Conformance Feature

A conformant processor MUST either be a conformant [schema-aware XSLT processor](#) or a conformant [non-schema-aware processor](#).

[DEFINITION: A **schema-aware XSLT processor** is an XSLT processor that implements the mandatory requirements of this specification connected with the [`xsl:import-schema`](#) declaration, the [`\[xsl:\]validation`](#) and [`\[xsl:\]type`](#) attributes, and the ability to handle input documents whose nodes have type annotations other than `xs:untyped` and `xs:untypedAtomic`. The mandatory requirements of this specification are taken to include the mandatory requirements of XPath 3.0, as described in [\[XPath 3.0\]](#). 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.]

[DEFINITION: A **non-schema-aware processor** is a processor that does not claim conformance with the schema-aware conformance feature. Such a processor MUST handle constructs associated with schema-aware processing as described in this section.]

[ERR XTSE1650] A [non-schema-aware processor](#) MUST signal a [static error](#) if a [package](#) 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 [3.14 Built-in Types](#).

A [non-schema-aware processor](#) is not able to validate input documents, and is not able to handle input documents containing type annotations other than `xs:untyped` or `xs:untypedAtomic`. Therefore, such a processor MUST treat any [`\[xsl:\]validation`](#) attribute with a value of `preserve` or `lax`, or a [`\[xsl:\]default-validation`](#) attribute with a value of `preserve` as if the value were `strip`.

Note:

The values `lax` and `preserve` indicate that the validation to be applied depends on the calling application, so it is appropriate for the request to be treated differently by different kinds of processor. By contrast, requesting `strict` validation, either through the [`\[xsl:\]validation`](#) attribute or the `type` attribute, indicates that the stylesheet is expecting to deal with typed data, and therefore cannot be processed without performing the validation.

[ERR XTSE1660] A [non-schema-aware processor](#) MUST signal a [static error](#) if a [package](#) includes an [`\[xsl:\]type`](#) attribute; or an [`\[xsl:\]validation`](#) or [`\[xsl:\]default-validation`](#) attribute with a value other than `strip`, `preserve`, or `lax`; or an [`xsl:mode`](#) element whose `typed` attribute is equal to `yes` or `strict`; or an `as` attribute whose value is a [SequenceType](#) that can only match nodes with a type annotation other than `xs:untyped` or `xs:untypedAtomic` (for example, `as="element(*, xs:integer)"`).

A [non-schema-aware processor](#) constrains the data model as follows, and raises a [dynamic error](#) ([see [ERR_XTDE1665](#)]) if the constraints are not satisfied:

- Atomic values MUST belong to one of the atomic types listed in [3.14 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`.

[27.3 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 [26 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.

Note:

A processor that does not claim conformance with the serialization feature MAY offer alternative serialization capabilities, and these MAY make use of the serialization parameters defined on [xsl:output](#) and/or [xsl:result-document](#).

If the processor claims conformance with the serialization feature then it MUST fully implement the [serialize](#)^{FO30} function defined in [\[Functions and Operators 3.0\]](#) or [\[Functions and Operators 3.1\]](#) as appropriate, and MUST NOT raise error [\[ERR_FODC0010\]](#)^{FO30} as the result of such a call.

If the processor does not claim conformance with the serialization feature, then it MAY raise error [\[ERR_FODC0010\]](#)^{FO30} in respect of some or all calls on the [serialize](#)^{FO30} function; it MUST NOT return a result from a call on this function unless the result is conformant with the specification, given the parameters actually supplied.

A processor that claims conformance with the Serialization Feature must satisfy the mandatory requirements of [\[XSLT and XQuery Serialization\]](#). It MUST provide a mode of operation which conforms to the 3.0 version of that specification. It MAY also provide a mode of operation which conforms to a later version of that specification; in

such cases the detail of how XSLT 3.0 interacts with new features introduced by such a version (for example, support for new serialization properties) is [implementation-defined](#).

[27.4 Compatibility Features](#)

[**DEFINITION:** A processor that claims conformance with the **XSLT 1.0 compatibility feature** **MUST** support the processing of stylesheet instructions and XPath expressions with [XSLT 1.0 behavior](#), as defined in [3.9 Backwards Compatible Processing](#).]

Note that a processor that does not claim conformance with the [XSLT 1.0 compatibility feature](#) **MUST** raise a [dynamic error](#) if an instruction is evaluated whose [effective version](#) is 1.0. [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, 2.0, or 3.0. 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 [XSLT 1.0 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.

Note:

There are no incompatibilities between 3.0 and 2.0 that would justify a 2.0-compatibility mode. When a 3.0 processor encounters a stylesheet that specifies `version="2.0"`, evaluation therefore proceeds exactly as if it specified `version="3.0"`. However, a software product may invoke an XSLT 2.0 processor in preference to an XSLT 3.0 processor when the stylesheet specifies `version="2.0"`, in which case any use of new 3.0 constructs will be rejected.

[27.5 Streaming Feature](#)

[**DEFINITION:** A processor that claims conformance with the **streaming feature** **MUST** use streamed processing in cases where (a) streaming is requested (for example by using the attribute `streamable="yes"` on [xsl:mode](#), or on the [xsl:source-document](#) instruction) and (b) the constructs in question are [guaranteed-streamable](#) according to this specification.]

A processor that does not claim conformance with the streaming feature is not required to use streamed processing and is not required to determine whether any construct is guaranteed streamable. Such a processor must, however, implement the semantics of all constructs in the language provided that enough memory is available to perform the processing without streaming.

A processor that conforms with the feature **MUST** return the value "yes" in response to the function call `system-property('xsl:supports-streaming')`; a processor that does not conform with the feature **MUST** return the value "no".

Note:

The term *streamed processing* as used here means the ability to process arbitrarily large input documents without ever-increasing memory requirements.

27.6 Dynamic Evaluation Feature

[**DEFINITION:** A processor that claims conformance with the **dynamic evaluation feature** MUST evaluate the [xsl:evaluate](#) function as described in this specification.]

A processor that does not claim conformance with the dynamic evaluation feature MUST report a dynamic error if an [xsl:evaluate](#) instruction is evaluated. It MUST NOT report a static error merely because of the presence of an [xsl:evaluate](#) instruction in the stylesheet, unless a processor that conforms with the feature would report the same static error.

A processor that conforms with the feature MUST return the value "yes" in response to the function call `system-property('xsl:supports-dynamic-evaluation')`; a processor that does not conform with the feature MUST return the value "no".

A processor that conforms with the feature MUST return the value `true` in response to the function call `element-available('xsl:evaluate')`; a processor that does not conform with the feature MUST return the value `false`.

Note:

A processor may allow dynamic evaluation to be enabled and disabled by means of configuration settings, perhaps for security reasons. In consequence, it may be impossible to tell during static analysis of the stylesheet whether or not the feature will be available during execution. A stylesheet author wanting to check whether the feature is available should therefore make the test using a run-time call on `system-property`, rather than relying on tests in an `[xsl:]use-when` attribute.

27.7 XPath 3.1 Feature

[**DEFINITION:** A processor that claims conformance with the **XPath 3.1 feature** MUST implement XPath 3.1 (including [\[XPath 3.1\]](#), [\[XDM 3.1\]](#), [\[XSLT and XQuery Serialization 3.1\]](#), and [\[Functions and Operators 3.1\]](#).)]

Specifically:

- All constructs where an [expression](#), [pattern](#), [SequenceType](#), or [ItemType](#) is expected must accept the XPath 3.1 grammar within those constructs.
- All functions defined in [\[Functions and Operators 3.1\]](#) are available.

Note:

Functions labeled as *higher-order* are available only if the [higher-order functions feature](#) is also available.

If both the XPath 3.1 feature and the Higher-Order Functions feature are available, then the [load-xquery-module^{FO31}](#) function will be available. However, as prescribed in the specification of that function, it has the option of returning a dynamic error if no suitable XQuery processor is available.

- The union type `xs:numeric` is recognized.

- The data model includes maps and arrays.

A processor that does not provide the XPath 3.1 feature constrains the data model by disallowing arrays, and may raise a [dynamic error](#) ([see [ERR_XTDE1665](#)]) if this constraint is not satisfied.

- Serialization of final results follows the rules in [\[XSLT and XQuery Serialization 3.1\]](#) (for example, it supports JSON serialization).
- The [xsl:evaluate](#) instruction supports dynamic evaluation of XPath 3.1 expressions.
- The result of `system-property("xsl>xpath-version")` is "3.1".

A processor that claims conformance with the XPath 3.1 feature MAY accept or reject constructs defined in any version of XPath (and its associated specifications) later than 3.1.

[27.7.1 Arrays](#)

XPath 3.1 introduces arrays as a new data structure, along with maps, largely in order to provide improved support for JSON. An array is an item, and it can be used as a function, so if \$A is an array, then \$A(3) selects the third member of the array, counting from one. The members of an array can be arbitrary values (that is, sequences).

Arrays become available in XSLT 3.0 when the [XPath 3.1 Feature](#) is implemented. There are no specific constructs in XSLT 3.0 to construct or manipulate arrays, but this can be achieved using facilities in XPath 3.1. The syntax for [SequenceTypes](#) is extended to allow arrays to be declared: for example `array(xs:integer)` represents an array whose members are (single) integers, while `array(map(xs:string, node()*))` represents an array whose members are maps from strings to sequences of nodes.

Like maps and sequences, arrays are immutable, and have no discernible identity (two arrays with the same members cannot be distinguished).

A number of functions for manipulating arrays are defined in [\[Functions and Operators 3.1\]](#).

[27.8 Higher-Order Functions Feature](#)

[**DEFINITION:** The **higher-order functions feature** contains functionality connected with the use of functions as items in the data model, that can be stored in variables and passed to other functions.]

[[ERR_XTSE3540](#)] A processor that does not provide the [higher-order functions feature](#) raises a [static error](#) if any of the following XPath constructs are found in an [expression](#), [pattern](#), [SequenceType](#), or [ItemType](#): a [TypedFunctionTest^{XP30}](#), a [NamedFunctionRef^{XP30}](#), an [InlineFunctionExpr^{XP30}](#), or an [ArgumentPlaceholder^{XP30}](#).

Note:

The effect is to disallow the three constructs used to create function-valued items: named function references such as `round#1`, inline function expressions such as `function($x){$x+1}`, and partial function application such as `starts-with(?, '#')`, along with sequence types such as `function(xs:integer) as xs:string` that serve no useful purpose in the absence of such items.

The item type `function(*)` is allowed by these rules, and serves as a generic type for maps and arrays.

Where a processor does not provide the [higher-order functions feature](#), functions whose specification in [\[Functions and Operators 3.1\]](#) labels them with the **higher-order** property are excluded from the static context of expressions and patterns. An attempt to reference such a function therefore fails in the same way as an attempt to call a non-existent function.

Note:

Examples of functions labeled with this property are [`filter`^{FO30}](#), [`for-each`^{FO30}](#), [`fold-left`^{FO30}](#), and [`fold-right`^{FO30}](#).

A processor that does not provide the higher-order functions feature constrains the data model by disallowing function items other than maps and arrays, and may raise a [dynamic error](#) ([see [ERR XTDE1665](#)]) if this constraint is not satisfied.

The same rules apply to a dynamic XPath expression processed using [`xsl:evaluate`](#).

A References

A.1 Normative References

XDM 3.0

[XQuery and XPath Data Model \(XDM\) 3.0](#), Norman Walsh, Anders Berglund, John Snelson, Editors. World Wide Web Consortium, 08 April 2014. This version is <https://www.w3.org/TR/2014/REC-xpath-datamodel-30-20140408/>. The [latest version](#) is available at <https://www.w3.org/TR/xpath-datamodel-30/>.

XDM 3.1

[XQuery and XPath Data Model \(XDM\) 3.1](#), Norman Walsh, John Snelson, Andrew Coleman, Editors. World Wide Web Consortium, 21 March 2017. This version is <https://www.w3.org/TR/2017/REC-xpath-datamodel-31-20170321/>. The [latest version](#) is available at <https://www.w3.org/TR/xpath-datamodel-31/>.

Functions and Operators 3.0

[XQuery and XPath Functions and Operators 3.0](#), Michael Kay, Editor. World Wide Web Consortium, 08 April 2014. This version is <https://www.w3.org/TR/2014/REC-xpath-functions-30-20140408/>. The [latest version](#) is available at <https://www.w3.org/TR/xpath-functions-30/>.

Functions and Operators 3.1

[XQuery and XPath Functions and Operators 3.1](#), Michael Kay, Editor. World Wide Web Consortium, 21 March 2017. This version is <https://www.w3.org/TR/2017/REC-xpath-functions-31-20170321/>. The [latest version](#) is available at <https://www.w3.org/TR/xpath-functions-31/>.

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 15924

ISO (International Organization for Standardization) *Information and documentation — Codes for the representation of names of scripts* ISO 15924:2004, January 2004. See <https://www.iso.org/obp/ui/#iso:std:iso:15924:ed-1:v1:en>.

ISO 15924 Register

Unicode Consortium. *Codes for the representation of names of scripts — Alphabetical list of four-letter script codes*. See <http://www.unicode.org/iso15924/iso15924-codes.html>. Retrieved February 2013; continually updated.

XSLT and XQuery Serialization

[XSLT and XQuery Serialization 3.0](#), Henry Zongaro, Andrew Coleman, Michael Sperberg-McQueen, Editors. World Wide Web Consortium, 08 April 2014. This version is <https://www.w3.org/TR/2014/REC-xslt-xquery-serialization-30-20140408/>. The [latest version](#) is available at <https://www.w3.org/TR/xslt-xquery-serialization-30/>.

XSLT and XQuery Serialization 3.1

[XSLT and XQuery Serialization 3.1](#), Andrew Coleman and Michael Sperberg-McQueen, Editors. World Wide Web Consortium, 21 March 2017. This version is <https://www.w3.org/TR/2017/REC-xslt-xquery-serialization-31-20170321/>. The [latest version](#) is available at <https://www.w3.org/TR/xslt-xquery-serialization-31/>.

RFC 7159

IETF. *The JavaScript Object Notation (JSON) Data Interchange Format*. March 2014. See <http://www.ietf.org/rfc/rfc7159.txt>

UNICODE

Unicode Consortium. *The Unicode Standard* as updated from time to time by the publication of new versions. See <http://www.unicode.org/standard/versions/> for the latest version and additional information on versions of the standard and of the Unicode Character Database. The version of Unicode to be used is [implementation-defined](#), but implementations are recommended to use the latest Unicode version.

UNICODE TR10

Unicode Consortium. *Unicode Technical Standard #10. Unicode Collation Algorithm*. Unicode Technical Report. See <http://www.unicode.org/reports/tr10/>.

UNICODE TR35

Unicode Consortium. *Unicode Technical Standard #35. Unicode Locale Data Markup Language*. Unicode Technical Report. See <http://www.unicode.org/reports/tr35/>.

XML 1.0

World Wide Web Consortium. *Extensible Markup Language (XML) 1.0. W3C Recommendation*. See <http://www.w3.org/TR/REC-xml/>. The edition of XML 1.0 must be no earlier than the Third Edition; the edition used is [implementation-defined](#), but we recommend that implementations use the latest version.

XML 1.1

[Extensible Markup Language \(XML\) 1.1 \(Second Edition\)](#), Tim Bray, Jean Paoli, Michael Sperberg-McQueen, 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 \(Second Edition\)](#), Jonathan Marsh and Richard Tobin, Editors. World Wide Web Consortium, 28 Jan 2009. This version is [http://www.w3.org/TR/2009/REC-xmlbase-20090128/](http://www.w3.org/TR/2009/REC-xmlbase-20090128). The [latest version](#) is

available at <http://www.w3.org/TR/xmlbase/>.

xml:id

[xml:id Version 1.0](#), Jonathan Marsh, Daniel Veillard, and Norman Walsh, 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

[Namespaces in XML 1.0 \(Third Edition\)](#), Tim Bray, Dave Hollander, Andrew Layman, *et. al.*, Editors. World Wide Web Consortium, 08 Dec 2009. This version is <http://www.w3.org/TR/2009/REC-xml-names-20091208/>. The [latest version](#) is available at <http://www.w3.org/TR/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 Thompson, David Beech, Murray Maloney, and Noah Mendelsohn, Editors. World Wide Web Consortium, 28 Oct 2004. This version is <http://www.w3.org/TR/2004/REC-xmleschema-1-20041028/>. The [latest version](#) is available at <http://www.w3.org/TR/xmleschema-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-xmleschema-2-20041028/>. The [latest version](#) is available at <http://www.w3.org/TR/xmleschema-2/>.

XML Schema 1.1 Part 1

[W3C XML Schema Definition Language \(XSD\) 1.1 Part 1: Structures](#), Sandy Gao, Michael Sperberg-McQueen, Henry Thompson, *et. al.*, Editors. World Wide Web Consortium, 05 Apr 2012. This version is <http://www.w3.org/TR/2012/REC-xmleschema11-1-20120405/>. The [latest version](#) is available at <http://www.w3.org/TR/xmleschema11-1/>.

XML Schema 1.1 Part 2

[W3C XML Schema Definition Language \(XSD\) 1.1 Part 2: Datatypes](#), David Peterson, Sandy Gao, Ashok Malhotra, *et. al.*, Editors. World Wide Web Consortium, 05 Apr 2012. This version is <http://www.w3.org/TR/2012/REC-xmleschema11-2-20120405/>. The [latest version](#) is available at <http://www.w3.org/TR/xmleschema11-2/>.

XPath 3.0

[XML Path Language \(XPath\) 3.0](#), Jonathan Robie, Don Chamberlin, Michael Dyck, John Snelsom, Editors. World Wide Web Consortium, 08 April 2014. This version is <https://www.w3.org/TR/2014/REC-xpath-30-20140408/>. The [latest version](#) is available at <https://www.w3.org/TR/xpath-30/>.

XPath 3.1

[XML Path Language \(XPath\) 3.1](#), Jonathan Robie, Michael Dyck and Josh Spiegel, Editors. World Wide Web Consortium, 21 March 2017. This version is <https://www.w3.org/TR/2017/REC-xpath-31-20170321/>. The [latest version](#) is available at <https://www.w3.org/TR/xpath-31/>.

XSLT Media Type

World Wide Web Consortium. *Registration of MIME Media Type application/xslt+xml*. In [Appendix B.1 of the XSLT 2.0 specification](#).

A.2 Other References

Unicode CLDR

CLDR - Unicode Common Locale Data Repository. Available at: <http://cldr.unicode.org>

DOM Level 2

Document Object Model (DOM) Level 2 Core Specification, Arnaud Le Hors, Philippe Le Hégaret, Lauren Wood, 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/>.

ECMA-404

ECMA International. *The JSON Data Interchange Format* October 2013. See <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>.

ICU

ICU - International Components for Unicode. Available at <http://site.icu-project.org>

RFC2119

S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels*. IETF RFC 2119. See <http://www.ietf.org/rfc/rfc2119.txt>.

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>.

RFC4647

A. Phillips and M. Davis. *Matching of Language Tags*. IETF RFC 4647. See <http://www.ietf.org/rfc/rfc4647.txt>.

RFC7303

H. Thompson and C. Lilley. *XML Media Types*. IETF RFC 7303. See <http://www.ietf.org/rfc/rfc7303.txt>

SemVer

Tom Preston-Werner, *Semantic Versioning 2.0.0*. See <http://semver.org/>. Undated (retrieved 1 August 2014).

STX

Petr Cimprich et al, *Streaming Transformations for XML (STX) Version 1.0*. Working Draft 27 April 2007. See <http://stx.sourceforge.net/documents/spec-stx-20070427.html>

XLink

XML Linking Language (XLink) Version 1.0, Steven DeRose, Eve Maler, and David Orchard, 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 <https://www.w3.org/TR/2005/NOTE-xml11schema10-20050511/>

XML Stylesheet

Associating Style Sheets with XML documents 1.0 (Second Edition), James Clark, Simon Pieters, and Henry Thompson, Editors. World Wide Web Consortium, 28 Oct 2010. This version is <http://www.w3.org/TR/2010/REC-xmlstylesheet-20101028>. The [latest version](#) is available at <http://www.w3.org/TR/xmlstylesheet>.

XPointer Framework

XPointer Framework, Paul Grosso, Eve Maler, Jonathan Marsh, 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/>.

XSL-FO

[*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/xsl11/>.

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

[*XSL Transformations \(XSLT\) Version 2.0 \(Second Edition\)*](#), Michael Kay, Editor. World Wide Web Consortium, 23 January 2007. This version is <https://www.w3.org/TR/2007/REC-xslt20-20070123/>. The [latest version](#) is available at [https://www.w3.org/TR/xslt20/](https://www.w3.org/TR/xslt20).

B XML Representation of JSON

This appendix contains the schema for the XML representation of JSON described in [22.1 XML Representation of JSON](#), together with the stylesheets used for converting from this XML representation to strings matching the JSON grammar.

These schema documents and stylesheets are also available as separate resources (links are listed at the top of this document).

B.1 Schema for the XML Representation of JSON

The schema is reproduced below:

```

<?xml version="1.0" encoding="UTF-8"?>
<xss:schema xmlns:xss="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  targetNamespace="http://www.w3.org/2005/xpath-functions"
  xmlns:j="http://www.w3.org/2005/xpath-functions">

  <!--
  * This is a schema for the XML representation of JSON used as the target for the
  * function fn:json-to-xml()
  *
  * The schema is made available under the terms of the W3C software notice and license
  * at http://www.w3.org/Consortium/Legal/copyright-software-19980720
  *
  -->

<xss:element name="map" type="j:mapType">
  <xss:unique name="unique-key">
    <xss:selector xpath="*"/>
    <xss:field xpath="@key"/>
    <xss:field xpath="@escaped-key"/>
  </xss:unique>
</xss:element>

<xss:element name="array" type="j:arrayType"/>

<xss:element name="string" type="j:stringType"/>

<xss:element name="number" type="j:numberType"/>

<xss:element name="boolean" type="xs:boolean"/>

<xss:element name="null" type="j:nullType"/>

<xss:complexType name="nullType">
  <xss:sequence/>
</xss:complexType>

<xss:complexType name="stringType">
  <xss:simpleContent>
    <xss:extension base="xs:string">
      <xss:attribute name="escaped" type="xs:boolean" use="optional" default="false"/>
    </xss:extension>
  </xss:simpleContent>
</xss:complexType>

<xss:simpleType name="numberType">
  <xss:restriction base="xs:double">
    <!-- exclude positive and negative infinity, and NaN -->
    <xss:minExclusive value="-INF"/>
    <xss:maxExclusive value="INF"/>
  </xss:restriction>
</xss:simpleType>

<xss:complexType name="arrayType">
  <xss:choice minOccurs="0" maxOccurs="unbounded">
    <xss:element ref="j:map"/>
    <xss:element ref="j:array"/>
    <xss:element ref="j:string"/>
    <xss:element ref="j:number"/>
    <xss:element ref="j:boolean"/>
    <xss:element ref="j:null"/>
  </xss:choice>
</xss:complexType>

```

```

</xs:choice>
</xs:complexType>

<xs:complexType name="mapType">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="map">
            <xs:complexType>
                <xs:complexContent>
                    <xs:extension base="j:mapType">
                        <xs:attribute name="key" type="xs:string"/>
                    </xs:extension>
                </xs:complexContent>
            </xs:complexType>
            <xs:unique name="unique-key-2">
                <xs:selector xpath="/" />
                <xs:field xpath="@key" />
            </xs:unique>
        </xs:element>
        <xs:element name="array">
            <xs:complexType>
                <xs:complexContent>
                    <xs:extension base="j:arrayType">
                        <xs:attributeGroup ref="j:key-group" />
                    </xs:extension>
                </xs:complexContent>
            </xs:complexType>
        </xs:element>
        <xs:element name="string">
            <xs:complexType>
                <xs:simpleContent>
                    <xs:extension base="j:stringType">
                        <xs:attributeGroup ref="j:key-group" />
                    </xs:extension>
                </xs:simpleContent>
            </xs:complexType>
        </xs:element>
        <xs:element name="number">
            <xs:complexType>
                <xs:simpleContent>
                    <xs:extension base="j:numberType">
                        <xs:attributeGroup ref="j:key-group" />
                    </xs:extension>
                </xs:simpleContent>
            </xs:complexType>
        </xs:element>
        <xs:element name="boolean">
            <xs:complexType>
                <xs:simpleContent>
                    <xs:extension base="xs:boolean">
                        <xs:attributeGroup ref="j:key-group" />
                    </xs:extension>
                </xs:simpleContent>
            </xs:complexType>
        </xs:element>
        <xs:element name="null">
            <xs:complexType>
                <xs:attributeGroup ref="j:key-group" />
            </xs:complexType>
        </xs:element>
    </xs:choice>
</xs:complexType>

```

```
<xss:attributeGroup name="key-group">
  <xss:attribute name="key" type="xs:string" use="required"/>
  <xss:attribute name="escaped-key" type="xs:boolean" use="optional" default="false"/>
</xss:attributeGroup>

</xss:schema>
```

B.2 Stylesheet for converting XML to JSON

This stylesheet contains the implementation of a function very similar to [xml-to-json](#), but implemented in XSLT so that it can be customized and extended. This stylesheet is provided for the benefit of users and there are no conformance requirements associated with it; there is no requirement that processors should make this stylesheet available. The stylesheet is reproduced below:

```

<?xml version="1.0" encoding="UTF-8"?>

<!--
  * This is a stylesheet for converting XML to JSON.
  * It expects the XML to be in the format produced by the XSLT 3.0 function
  * fn:json-to-xml(), but is designed to be highly customizable.
  *
  * The stylesheet is made available under the terms of the W3C software notice and license
  * at http://www.w3.org/Consortium/Legal/copyright-software-19980720
  *
-->

<xsl:package
  name="http://www.w3.org/2013/XSLT/xml-to-json"
  package-version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:fn="http://www.w3.org/2005/xpath-functions"
  xmlns:j="http://www.w3.org/2013/XSLT/xml-to-json"
  exclude-result-prefixes="xs fn j" default-mode="j:xml-to-json" version="3.0">

  <xsl:variable name="quot" visibility="private"></xsl:variable>
  <xsl:param name="indent-spaces" select="2"/>

  <!-- The static parameter STREAMABLE controls whether the stylesheet is declared as streamable -->
  <xsl:param name="STREAMABLE" static="yes" as="xs:boolean" select="true()"/>

  <xsl:mode name="indent" _streamable="{$STREAMABLE}" visibility="public"/>
  <xsl:mode name="no-indent" _streamable="{$STREAMABLE}" visibility="public"/>
  <xsl:mode name="key-attribute" streamable="false" on-no-match="fail" visibility="public"/>

  <!-- The static parameter VALIDATE controls whether the input, if untyped, should be validated -->
  <xsl:param name="VALIDATE" static="yes" as="xs:boolean" select="false()"/>
  <xsl:import-schema namespace="http://www.w3.org/2005/xpath-functions" use-when="$VALIDATE"/>

  <!-- Entry point: function to convert a supplied XML node to a JSON string -->
  <xsl:function name="j:xml-to-json" as="xs:string" visibility="public">
    <xsl:param name="input" as="node()"/>
    <xsl:sequence select="j:xml-to-json($input, map{})"/>
  </xsl:function>

  <!-- Entry point: function to convert a supplied XML node to a JSON string, supplying options -->
  <xsl:function name="j:xml-to-json" as="xs:string" visibility="public">
    <xsl:param name="input" as="node()"/>
    <xsl:param name="options" as="map(*)"/>
    <xsl:variable name="input" as="node()" use-when="$VALIDATE">
      <xsl:copy-of select="$input" validation="strict"/>
    </xsl:variable>
    <xsl:choose>
      <xsl:when test="$options('indent') eq true()">
        <xsl:apply-templates select="$input" mode="indent">
          <xsl:with-param name="fallback" as="(function(element()) as xs:string)?"
            select="$options('fallback')" tunnel="yes"/>
        </xsl:apply-templates>
      </xsl:when>
      <xsl:otherwise>
        <xsl:apply-templates select="$input" mode="no-indent">
          <xsl:with-param name="fallback" as="(function(element()) as xs:string)?"
            select="$options('fallback')" tunnel="yes"/>
        </xsl:apply-templates>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:function>
</xsl:package>

```

```

        select="$options('fallback')" tunnel="yes"/>
    </xsl:apply-templates>
</xsl:otherwise>
</xsl:choose>
</xsl:function>

<!-- A document node is ignored -->

<xsl:template match="/" mode="indent no-indent">
    <xsl:apply-templates mode="#current"/>
</xsl:template>

<!-- Template rule for fn:map elements, representing JSON objects -->

<xsl:template match="fn:map" mode="indent">
    <xsl:value-of>
        <xsl:variable name="depth" select="count(ancestor::*) + 1"/>
        <xsl:text>{</xsl:text>
        <xsl:for-each select="*"
            <xsl:if test="position() gt 1">
                <xsl:text>, </xsl:text>
                <xsl:value-of select="j:indent($depth)"/>
            </xsl:if>
            <xsl:apply-templates select="snapshot(@key)" mode="key-attribute"/>
            <xsl:text> : </xsl:text>
            <xsl:apply-templates select=". ." mode="#current"/>
        </xsl:for-each>
        <xsl:text>}</xsl:text>
    </xsl:value-of>
</xsl:template>

<xsl:template match="fn:map" mode="no-indent">
    <xsl:value-of>
        <xsl:text>{</xsl:text>
        <xsl:for-each select="*"
            <xsl:if test="position() gt 1">
                <xsl:text>, </xsl:text>
            </xsl:if>
            <xsl:apply-templates select="snapshot(@key)" mode="key-attribute"/>
            <xsl:text>:</xsl:text>
            <xsl:apply-templates select=". ." mode="#current"/>
        </xsl:for-each>
        <xsl:text>}</xsl:text>
    </xsl:value-of>
</xsl:template>

<!-- Template rule for fn:array elements, representing JSON arrays -->
<xsl:template match="fn:array" mode="indent">
    <xsl:value-of>
        <xsl:variable name="depth" select="count(ancestor::*) + 1"/>
        <xsl:text>[</xsl:text>
        <xsl:for-each select="*"
            <xsl:if test="position() gt 1">
                <xsl:text>, </xsl:text>
                <xsl:value-of select="j:indent($depth)"/>
            </xsl:if>
            <xsl:apply-templates select=". ." mode="#current"/>
        </xsl:for-each>
        <xsl:text>]</xsl:text>
    </xsl:value-of>
</xsl:template>
```

```

<xsl:template match="fn:array" mode="no-indent">
  <xsl:value-of>
    <xsl:text>[</xsl:text>
    <xsl:for-each select="*"
      <xsl:if test="position() gt 1">
        <xsl:text>,</xsl:text>
      </xsl:if>
      <xsl:apply-templates select=".." mode="#current"/>
    </xsl:for-each>
    <xsl:text>]</xsl:text>
  </xsl:value-of>
</xsl:template>

<!-- Template rule for fn:string elements in which
     special characters are already escaped -->
<xsl:template match="fn:string[@escaped='true']" mode="indent no-indent">
  <xsl:sequence select="concat($quot, .., $quot)" />
</xsl:template>

<!-- Template rule for fn:string elements in which
     special characters need to be escaped -->
<xsl:template match="fn:string[not(@escaped='true')]" mode="indent no-indent">
  <xsl:sequence select="concat($quot, j:escape(.), $quot)" />
</xsl:template>

<!-- Template rule for fn:boolean elements -->
<xsl:template match="fn:boolean" mode="indent no-indent">
  <xsl:sequence select="xs:string(xs:boolean(.))" />
</xsl:template>

<!-- Template rule for fn:number elements -->
<xsl:template match="fn:number" mode="indent no-indent">
  <xsl:value-of select="xs:string(xs:double(.))" />
</xsl:template>

<!-- Template rule for JSON null elements -->
<xsl:template match="fn:null" mode="indent no-indent">
  <xsl:text>null</xsl:text>
</xsl:template>

<!-- Template rule matching a key within a map where
     special characters in the key are already escaped -->
<xsl:template match="fn:/*[@key-escaped='true']/@key" mode="key-attribute">
  <xsl:value-of select="concat($quot, .., $quot)" />
</xsl:template>

<!-- Template rule matching a key within a map where
     special characters in the key need to be escaped -->
<xsl:template match="fn:/*[not(@key-escaped='true')]/@key" mode="key-attribute">
  <xsl:value-of select="concat($quot, j:escape(.), $quot)" />
</xsl:template>

<!-- Template matching "invalid" elements -->
<xsl:template match="*" mode="indent no-indent">
  <xsl:param name="fallback" as="(function(element()) as xs:string)?"
    tunnel="yes" required="yes" />
  <xsl:choose>
    <xsl:when test="exists($fallback)">
      <xsl:value-of select="$fallback(snapshot(.))" />
    </xsl:when>
  </xsl:choose>
</xsl:template>

```

```

<xsl:otherwise>
    <xsl:message terminate="yes">Inc</xsl:message>
</xsl:otherwise>
</xsl:choose>
</xsl:template>

<!-- Template rule matching (and discarding) whitespace text nodes in the XML --&gt;
&lt;xsl:template match="text()[not(normalize-space())]" mode="indent no-indent"/&gt;

<!-- Function to escape special characters --&gt;
&lt;xsl:function name="j:escape" as="xs:string" visibility="final"&gt;
    &lt;xsl:param name="in" as="xs:string"/&gt;
    &lt;xsl:value-of&gt;
        &lt;xsl:for-each select="string-to-codepoints($in)"&gt;
            &lt;xsl:choose&gt;
                &lt;xsl:when test=". &gt;t 65535"&gt;
                    &lt;xsl:value-of select="concat('\u', j:hex4(. - 65536) idiv 1024 + 55296
                        &lt;xsl:value-of select="concat('\u', j:hex4(. - 65536) mod 1024 + 56320)
                &lt;/xsl:when&gt;
                &lt;xsl:when test=". = 34"&gt;\&lt;/xsl:when&gt;
                &lt;xsl:when test=". = 92"&gt;\&lt;/xsl:when&gt;
                &lt;xsl:when test=". = 08"&gt;\b&lt;/xsl:when&gt;
                &lt;xsl:when test=". = 09"&gt;\t&lt;/xsl:when&gt;
                &lt;xsl:when test=". = 10"&gt;\n&lt;/xsl:when&gt;
                &lt;xsl:when test=". = 12"&gt;\f&lt;/xsl:when&gt;
                &lt;xsl:when test=". = 13"&gt;\r&lt;/xsl:when&gt;
                &lt;xsl:when test=". lt 32 or (. ge 127 and . le 160)"&gt;
                    &lt;xsl:value-of select="concat('\u', j:hex4(.))"/&gt;
                &lt;/xsl:when&gt;
                &lt;xsl:otherwise&gt;
                    &lt;xsl:value-of select="codepoints-to-string(.)"/&gt;
                &lt;/xsl:otherwise&gt;
            &lt;/xsl:choose&gt;
        &lt;/xsl:for-each&gt;
    &lt;/xsl:value-of&gt;
&lt;/xsl:function&gt;

<!-- Function to convert a UTF16 codepoint into a string of four hex digits --&gt;
&lt;xsl:function name="j:hex4" as="xs:string" visibility="final"&gt;
    &lt;xsl:param name="ch" as="xs:integer"/&gt;
    &lt;xsl:variable name="hex" select="'0123456789abcdef'"/&gt;
    &lt;xsl:value-of&gt;
        &lt;xsl:value-of select="substring($hex, $ch idiv 4096 + 1, 1)"/&gt;
        &lt;xsl:value-of select="substring($hex, $ch idiv 256 mod 16 + 1, 1)"/&gt;
        &lt;xsl:value-of select="substring($hex, $ch idiv 16 mod 16 + 1, 1)"/&gt;
        &lt;xsl:value-of select="substring($hex, $ch mod 16 + 1, 1)"/&gt;
    &lt;/xsl:value-of&gt;
&lt;/xsl:function&gt;

<!-- Function to output whitespace indentation based on
     the depth of the node supplied as a parameter --&gt;
&lt;xsl:function name="j:indent" as="text()" visibility="public"&gt;
    &lt;xsl:param name="depth" as="xs:integer"/&gt;
    &lt;xsl:value-of select="'\xa;', string-join((1 to ($depth + 1) * $indent-spaces) ! ' ',&gt;
&lt;/xsl:function&gt;

&lt;/xsl:package&gt;
</pre>

```

C Glossary (Non-Normative)

absent

A component of the context that has no value is said to be **absent**.

absorption

An operand usage of **absorption** indicates that the construct reads the subtree(s) rooted at a supplied node(s).

accumulator

An **accumulator** defines a series of values associated with the nodes of the tree. If an accumulator is applicable to a particular tree, then for each node in the tree, other than attribute and namespace nodes, there will be two values available, called the pre-descent and post-descent values. These two values are available via a pair of functions, [accumulator-before](#) and [accumulator-after](#).

accumulator function

The functions [accumulator-before](#) and [accumulator-after](#) are referred to as the **accumulator functions**.

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](#).

applicable

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 specified in the [xsl:]default-mode attribute of the innermost containing element that has such an attribute, which in turn defaults to the [unnamed mode](#). If the mode 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.

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](#)^{XP30}. It is a process that takes as input a sequence of items, and returns a sequence of atomic values, in which the nodes are replaced by their typed values as defined in [\[XDM 3.0\]](#). If the [XPath 3.1 Feature](#) is implemented, then arrays (see [27.7.1 Arrays](#)) are atomized by atomizing their members, recursively.

attribute set

An **attribute set** is defined as a set of [xsl:attribute-set](#) declarations in the same [package](#) that share the same [expanded QName](#).

attribute set invocation

An **attribute set invocation** is a pseudo-instruction corresponding to a single EQName appearing within an [xsl:]use-attribute-sets attribute; the effect of the pseudo-instruction is to cause the referenced [attribute set](#) to be evaluated.

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 ({}), following the general rules for [value templates](#).

backwards compatible behavior

An element is processed with **backwards compatible behavior** if its [effective version](#) is less than 3.0.

base output URI

The **base output URI** is a URI to be used as the base URI when resolving a relative URI reference 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 constructs explicitly associated with an optional feature.

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.

choice operand group

For some construct kinds, one or more operand roles may be defined to form a **choice operand group**. This concept is used where it is known that [operands](#) are mutually exclusive (for example the `then` and `else` clauses in a conditional expression).

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.

climbing

Climbing: indicates that streamed nodes returned by the construct are reached by navigating the parent, ancestor[-or-self], attribute, and/or namespace axes from the node at the current streaming position.

collation

Facilities in XSLT 3.0 and XPath 3.0 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.

combined posture

The **combined posture** of a [choice operand group](#) is determined by the [postures](#) of the [operands](#) in the group (the **operand postures**), and is the first of the following that applies:

1. If any of the operand postures is [roaming](#), then the combined posture is [roaming](#).
2. If all of the operand postures are [grounded](#), then the combined posture is [grounded](#).
3. If one or more of the operand postures is [climbing](#) and the remainder (if any) are [grounded](#), then the combined posture is [climbing](#).
4. If one or more of the operand postures is [striding](#) and the remainder (if any) are [grounded](#), then the combined posture is [striding](#).
5. If one or more of the operand postures is [crawling](#) and each of the remainder (if any) is either [striding](#) or [grounded](#), then the combined posture is [crawling](#).
6. Otherwise (for example, if the group includes both an operand with [climbing](#) posture and one with [crawling](#) posture), the combined posture is [roaming](#).

compatible

The signatures of two [components](#) are **compatible** if they present the same interface to the user of the component. The additional rules depend on the kind of component.

[component](#)

The term **component** is used to refer to any of the following: a [stylesheet function](#), a [named template](#), a [mode](#), an [accumulator](#), an [attribute set](#), a [key](#), [global variable](#), or a [mode](#).

[composite merge key value](#)

The ordered collection of [merge key values](#) computed for one item in a [merge input sequence](#) (one for each [merge key component](#) within the [merge key specification](#)) is referred to as a **composite merge key value**.

[construct](#)

The term **construct** refers to the union of the following: a [sequence constructor](#), an [instruction](#), an [attribute set](#), a [value template](#), an [expression](#), or a [pattern](#).

[consuming](#)

A **consuming** construct is any [construct](#) deemed consuming by the rules in this section ([19 Streamability](#)).

[containing package](#)

A component declaration results in multiple components, one in the package in which the declaration appears, and potentially one in each package that uses the declaring package, directly or indirectly, subject to the visibility of the component. Each of these multiple components has the same [declaring package](#), but each has a different **containing package**. For the original component, the declaring package and the containing package are the same; for a copy of a component made as a result of an [xsl:use-package](#) declaration, the declaring package will be the original package, and the containing package will be the package in which the [xsl:use-package](#) declaration appears.

[context item](#)

The **context item** is the item currently being processed. An item (see [\[XDM 3.0\]](#)) is either an atomic value (such as an integer, date, or string), a node, or a function item. 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 item type](#)

For every expression, it is possible to establish by static analysis, information about the item type of the context item for evaluation of that expression. This is called the **context item type** of the expression.

[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 or a function item, 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 posture](#)

The **context posture**. This captures information about how the [context item](#) used as input to the construct is positioned relative to the streamed input. The **context posture** of a construct C is the posture of the expression whose value sets the focus for the evaluation of C.

[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).

controlled operand

Within a [focus-changing construct](#) there are one or more [operands](#) that are evaluated with a [focus](#) determined by the [controlling operand](#) (or in some cases such as [xsl:on-completion](#), with an [absent focus](#)); these are referred to as **controlled operands**.

controlling operand

Within a [focus-changing construct](#) there is in many cases one [operand](#) whose value determines the [focus](#) for evaluating other operands; this is referred to as the **controlling operand**.

crawling

Crawling: typically indicates that streamed nodes returned by a construct are reached by navigating the descendant[-or-self] axis.

current captured substrings

While the [xsl:matching-substring](#) instruction is active, a set of **current captured substrings** is available, corresponding to the parenthesized subexpressions of the regular expression.

current group

The **current group** is the [group](#) itself, as a sequence of items

current grouping key

The **current grouping key** is a single atomic value, or in the case of a composite key, a sequence of atomic values, containing the [grouping key](#) of the items in the [current group](#).

current merge group

The **current merge group** is a [map](#). During evaluation of an [xsl:merge](#) instruction, as each group of items with equal [composite merge key values](#) is processed, the current merge group is set to a map whose keys are the names of the various merge sources, and whose associated values are the items from each merge source having the relevant composite merge key value.

current merge key

The **current merge key** is a sequence of atomic values. During evaluation of an [xsl:merge](#) instruction, as each group of items with equal [composite merge key values](#) is processed, the current merge key is set to the composite merge key value that these items have in common.

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 [initial mode](#), 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 output URI

The **current output URI** is the URI associated with the [principal result](#) or [secondary result](#) that is currently being written.

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.

decimal format

All the xsl:decimal-format declarations in a package 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 [3.7.3 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.

declared-streamable

The above constructs (template rules belonging to a mode declared with streamable="yes"; and xsl:source-document, xsl:attribute-set, xsl:function, xsl:merge-source, and xsl:accumulator elements specifying streamable="yes") are said to be **declared-streamable**.

declaring package

The **declaring package** of a component is the package that contains the declaration (or, in the case of xsl:attribute-set and xsl:key, multiple declarations) of the component.

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 priority

If no priority attribute is specified on an 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 an attribute or text node within a stylesheet module, or as the result of evaluating an XPath expression contained in such a node. The element containing this attribute or text node is referred to as the **defining element** of the lexical 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 capable of detection until a source document is being transformed is referred to as a **dynamic error**.

dynamic evaluation feature

A processor that claims conformance with the **dynamic evaluation feature** MUST evaluate the xsl:evaluate function as described in this specification.

effective value

The result of evaluating a value template is referred to as its **effective value**.

effective version

The **effective version** of an element in a [stylesheet module](#) or [package manifest](#) is the decimal value of the `[xsl:]version` attribute (see [3.4 Standard Attributes](#)) on that element or on the innermost ancestor element that has such an attribute, excluding the `version` attribute on an [xsl:output](#) element.

embedded stylesheet module

A stylesheet module whose outermost element is the child of a non-XSLT element in a host document is referred to as an **embedded stylesheet module**. See [3.12 Embedded Stylesheet Modules](#).

EQName

An **EQName** is a string representing an [expanded QName](#) where the string, after removing leading and trailing whitespace, is in the form defined by the [EQName^{XP30}](#) production in the XPath specification.

expanded QName

An **expanded QName** is a value in the value space of the `xs:QName` datatype as defined in the XDM data model (see [\[XDM 3.0\]](#)): that is, a triple containing namespace prefix (optional), namespace URI (optional), and local name. 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.

explicit default

An **explicit default** for a parameter is indicated by the presence of either a `select` attribute or a non-empty sequence constructor.

explicitly mandatory

A parameter is **explicitly mandatory** if it is a [function parameter](#), or if the `required` attribute is present and has the value `yes`.

expression

Within this specification, the term **XPath expression**, or simply **expression**, means a string that matches the production [Expr^{XP30}](#) defined in [\[XPath 3.0\]](#), with the extensions defined in [21 Maps](#).

extension attribute

An element from the XSLT namespace may have any attribute not from the XSLT namespace, provided that the [expanded QName](#) (see [\[XPath 3.0\]](#)) of the attribute has a non-null namespace URI. These attributes are referred to as **extension attributes**.

extension function

An **extension function** is a named function introduced to the static or dynamic context by mechanisms outside the scope of this specification.

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 output of a transformation: specifically, a tree built by post-processing the items in the [principal result](#) or in a [secondary result](#). 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**.

[focus-changing construct](#)

A **focus-changing construct** is a [construct](#) that has one or more [operands](#) that are evaluated with a different [focus](#) from the parent construct.

[focus-setting container](#)

The **focus-setting container** of a construct C is the innermost [focus-changing construct](#) F (if one exists) such that C is directly or indirectly contained in a [controlled operand](#) of F . If there is no such construct F , then the focus-setting container is the containing [declaration](#), for example an [xsl:function](#) or [xsl:template](#) element.

[forwards compatible behavior](#)

An element is processed with **forwards compatible behavior** if its [effective version](#) is greater than 3.0.

[free-ranging](#)

A **free-ranging** construct is any [construct](#) deemed free-ranging by the rules in this section ([19 Streamability](#)).

[function conversion rules](#)

When used in this specification without further qualification, the term **function conversion rules** means the function conversion rules defined in [\[XPath 3.0\]](#), applied with XPath 1.0 compatibility mode 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](#).

[fundamental item type](#)

There are 28 **fundamental item types**: the 7 node kinds defined in [\[XDM 3.0\]](#) (element, attribute, etc.), the 19 primitive atomic types defined in [\[XML Schema Part 2\]](#), plus the types `function(*)` and `xs:untypedAtomic`. The fundamental item types are disjoint, and every item is an instance of exactly one of them.

[general streamability rules](#)

Many [constructs](#) share the same streamability rules. These rules, referred to as the **general streamability rules**, are defined here.

[global context item](#)

An item that acts as the **global context item** for the transformation. This item acts as the [context item](#) when evaluating the [select expression](#) or [sequence constructor](#) of a [global variable](#) declaration within the [top-level package](#), as described in [5.3.3.1 Maintaining Position: the Focus](#). The global context item may also be available in a [named template](#) when the stylesheet is invoked as described in [2.3.4 Call-Template Invocation](#).

global variable

A [top-level variable-binding element](#) declares a **global variable** that is visible everywhere (except within its own declaration, and where it is [shadowed](#) by another binding).

grounded

Grounded: indicates that the value returned by the construct does not contain nodes from the streamed input document

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 item in a group must match.

grouping key

If either of the group-by or group-adjacent attributes is present, then for each item in the [population](#) a set of **grouping keys** is calculated, as follows: the expression contained in the group-by or group-adjacent attribute is evaluated; the result is atomized; and any `xs:untypedAtomic` values are cast to `xs:string`. If composite="yes" is specified, there is a single grouping key whose value is the resulting sequence; otherwise, there is a set of grouping keys, consisting of the distinct atomic values present in the result sequence.

guaranteed-streamable

A **guaranteed-streamable** construct is a [construct](#) that is declared to be streamable and that follows the particular rules for that construct to make streaming possible, as defined by the analysis in this specification.

higher-order functions feature

The **higher-order functions feature** contains functionality connected with the use of functions as items in the data model, that can be stored in variables and passed to other functions.

higher-order operand

Whether or not the [operand](#) is **higher-order**. For this purpose an operand O of a construct C is higher-order if the semantics of C potentially require O to be evaluated more than once during a single evaluation of C .

homonymous

Two [components](#) are said to be **homonymous** if they have the same [symbolic identifier](#).

identical (types)

Types S and T are considered **identical** for the purpose of these rules if and only if $\text{subtype}(S, T)$ and $\text{subtype}(T, S)$ both hold, where the subtype relation is defined in [Section 2.5.6.1 The judgement subtype\(A, B\)](#)^{XP30}.

immediate result

The result of evaluating a [sequence constructor](#) is the sequence of items formed by concatenating the results of evaluating each of the nodes in the sequence constructor, retaining order. This is referred to as the **immediate result** of the sequence constructor.

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.

implicit default

If a parameter that is not explicitly mandatory has no explicit default value, then it has an **implicit default** value, which is the empty sequence if there is an `as` attribute, or a zero-length string if not.

implicitly mandatory

If a parameter has an implicit default value which cannot be converted to the required type (that is, if it has an `as` attribute which does not permit the empty sequence), then the parameter is **implicitly mandatory**.

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 package are referred to as the **in-scope schema components**.

initial function

A stylesheet may be evaluated by calling a named stylesheet function, referred to as the **initial function**.

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 match selection

A stylesheet may be evaluated by supplying a value to be processed, together with an initial mode. The value (which can be any sequence of items) is referred to as the **initial match selection**. The processing then corresponds to the effect of the `xsl:apply-templates` instruction.

initial mode

The **initial mode** is the mode used to select template rules for processing items in the initial match selection when `apply-templates` invocation is used to initiate a transformation.

initial named template

A stylesheet may be evaluated by selecting a named template to be evaluated; this is referred to as the **initial named template**.

initial sequence

The sequence to be sorted is referred to as the **initial sequence**.

initial setting

The **initial setting** of a component of the dynamic context is used when evaluating global variables and stylesheet parameters, when evaluating the `use` and `match` attributes of `xsl:key`, and when evaluating the `initial-value` of `xsl:accumulator` and the `select` expressions or contained sequence constructors of `xsl:accumulator-rule`.

inspection

An operand usage of **inspection** indicates that the construct accesses properties of a supplied node that are available without reading its subtree.

instruction

An **instruction** is either an [XSLT instruction](#) or an [extension instruction](#).

invocation construct

The following [constructs](#) are classified as **invocation constructs**: the instructions [xsl:call-template](#), [xsl:apply-templates](#), [xsl:apply-imports](#), and [xsl:next-match](#); XPath function calls that bind to [stylesheet functions](#); XPath dynamic function calls; the functions [accumulator-before](#) and [accumulator-after](#); the [xsl:]use-attribute-sets attribute. These all have the characteristic that they can cause evaluation of constructs that are not lexically contained within the calling construct.

key

A **key** is defined as a set of [xsl:key](#) declarations in the same [package](#) 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 an [expanded QName](#) where the string, after removing leading and trailing whitespace, is within the lexical space of the [xs:QName](#) datatype as defined in XML Schema (see [\[XML Schema Part 2\]](#)): that is, a local name optionally preceded by a namespace prefix and a colon.

library package

Every [package](#) within a [stylesheet](#), other than the [top-level package](#), is referred to as a **library package**.

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 [24.2 Extension Instructions](#)) is classified as a **literal result element**.

local variable

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

map

A map consists of a set of entries. Each entry comprises a key which is an arbitrary atomic value, and an arbitrary sequence called the associated value.

match type

The **match type** of a [pattern](#) is the most specific [U-type](#) that is known to match all items that the pattern can match.

merge activation

A **merge activation** is a single evaluation of the sequence constructor contained within the [xsl:merge-action](#) element, which occurs once for each distinct [composite merge key value](#).

merge input sequence

A **merge input sequence** is an arbitrary [sequence^{DM30}](#) of items which is already sorted according to the [merge key specification](#) for the corresponding [merge source definition](#).

[merge key component](#)

A **merge key component** specifies one component of a [merge key specification](#); it corresponds to a single [`xsl:merge-key`](#) element in the stylesheet.

[merge key specification](#)

A **merge key specification** consists of one or more adjacent [`xsl:merge-key`](#) elements which together define how the [merge input sequences](#) selected by a [merge source definition](#) are sorted. Each [`xsl:merge-key`](#) element defines one [merge key component](#).

[merge key value](#)

For each item in a [merge input sequence](#), a value is computed for each [merge key component](#) within the [merge key specification](#). The value computed for an item by using the *N*th [merge key component](#) is referred to as the *N*th [merge key value](#) of that item.

[merge source definition](#)

A **merge source definition** is the definition of one kind of input to the merge operation. It selects zero or more [merge input sequences](#), and it includes a [merge key specification](#) to define how the [merge key values](#) are computed for each such merge input sequence.

[mode](#)

A **mode** is a set of template rules; when the [`xsl:apply-templates`](#) instruction selects a set of items for processing, it identifies the rules to be used for processing those items by nominating a mode, explicitly or implicitly.

[mode definition](#)

All the [`xsl:mode`](#) declarations in a [package](#) that share the same name are grouped into a named **mode definition**; those that have no name are grouped into a single unnamed mode definition.

[motionless](#)

A **motionless** construct is any [construct](#) deemed motionless by the rules in this section ([19 Streamability](#)).

[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**.

[navigation](#)

An operand usage of **navigation** indicates that the construct may navigate freely from the supplied node to other nodes in the same tree, in a way that is not constrained by the streamability rules.

[non-contextual function call](#)

The term **non-contextual function call** is used to refer to function calls that do not pass the dynamic context to the called function. This includes all calls on [stylesheet functions](#) and all [dynamic function invocations^{XP30}](#), (that is calls to function items as permitted by XPath 3.0). It excludes calls to some functions in the namespace <http://www.w3.org/2005/xpath-functions>, in particular those that explicitly depend on the context,

such as the [current-group](#) and [regex-group](#) functions. It is [implementation-defined](#) whether, and under what circumstances, calls to [extension functions](#) are non-contextual.

[non-positional predicate](#)

A predicate is a **non-positional predicate** if it satisfies both of the following conditions:

1. The predicate does not contain a function call or named function reference to any of the following functions, unless that call or reference occurs within a nested predicate:

- a. [position](#)^{FO30}
- b. [last](#)^{FO30}
- c. [function-lookup](#)^{FO30}.

Note:

The exception for nested predicates is there to ensure that patterns such as `match="p[@code = $status[last()]]` are not disqualified.

2. The expression immediately contained in the predicate is a non-numeric expression. An expression is non-numeric if the intersection of its [static type](#) (see [19.1 Determining the Static Type of a Construct](#)) with $U\{xs:decimal, xs:double, xs:float\}$ is $U\{\}$.

[non-schema-aware processor](#)

A **non-schema-aware processor** is a processor that does not claim conformance with the schema-aware conformance feature. Such a processor **MUST** handle constructs associated with schema-aware processing as described in this section.

[operand](#)

In an actual instance of a construct, there will be a number of **operands**. Each operand is itself a [construct](#); the construct tree can be defined as the transitive relation between constructs and their operands.

[operand role](#)

For every construct kind, there is a set of zero or more **operand roles**.

[operand usage](#)

The **operand usage**. This gives information, in the case where the operand value contains nodes, about how those nodes are used. The operand usage takes one of the values [absorption](#), [inspection](#), [transmission](#), or [navigation](#).

[option parameter conventions](#)

Functions that take an options parameter adopt common conventions on how the options are used. These are referred to as the **option parameter conventions**. These rules apply only to functions that explicitly refer to them.

[order of first appearance](#)

There is a total 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 within a [package](#) 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](#)

[override](#)

A component in a using package may **override** a component in a used package, provided that the [visibility](#) of the component in the used package is either [abstract](#) or [public](#). The overriding declaration is written as a child of the [xsl:override](#) element, which in turn appears as a child of [xsl:use-package](#).

[package](#)

An explicit **package** is represented by an [xsl:package](#) element, which will generally be the outermost element of an XML document. When the [xsl:package](#) element is not used explicitly, the entire stylesheet comprises a single implicit package.

[package manifest](#)

The content of the [xsl:package](#) element is referred to as the **package manifest**

[parameter](#)

The [xsl:param](#) element declares a **parameter**, which may be a [stylesheet parameter](#), a [template parameter](#), a [function parameter](#), or an [xsl:iterate](#) parameter. A parameter is a [variable](#) with the additional property that its value can be set by the caller.

[pattern](#)

A **pattern** specifies a set of conditions on an item. An item that satisfies the conditions matches the pattern; an item that does not satisfy the conditions does not match the pattern.

[picture string](#)

The **picture string** is the string supplied as the second argument of the [format-number](#)^{FO30} function.

[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**

[portion](#)

The integer literals and the optional [NamePart](#) within the version number are referred to as the **portions** of the version number.

[posture](#)

The **posture** of the expression. This captures information about the way in which the streamed input document is positioned on return from evaluating the construct. The posture takes one of the values [climbing](#), [striding](#), [crawling](#), [roaming](#), or [grounded](#).

[potentially consuming](#)

An [operand](#) is **potentially consuming** if at least one of the following conditions applies:

- i. The operand's adjusted [sweep](#) S' is [consuming](#).
- ii. The [operand usage](#) is [transmission](#) and the operand is not [grounded](#).

[predicate pattern](#)

A **predicate pattern** is written as . (dot) followed by zero or more predicates in square brackets, and it matches any item for which each of the predicates evaluates to `true`.

[principal result](#)

A **principal result**: this can be any sequence of items (as defined in [\[XDM 3.0\]](#)).

[principal stylesheet module](#)

Within a [package](#), one [stylesheet module](#) functions as the **principal stylesheet module**. The complete package is assembled by finding the stylesheet modules referenced directly or indirectly from the principal stylesheet module using [xsl:include](#) and [xsl:import](#) elements: see [3.11.2 Stylesheet Inclusion](#) and [3.11.3 Stylesheet Import](#).

[priority](#)

The **priority** of a template rule is specified by the `priority` attribute on the [xsl:template](#) declaration. If no priority is specified explicitly for a template rule, its [default priority](#) is used, as defined in [6.5 Default Priority for Template Rules](#).

[processing order](#)

There is another total 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.

[raw result](#)

The result of invoking the selected component, after any required conversion to the declared result type of the component, is referred to as the **raw result**.

[reference binding](#)

The process of identifying the [component](#) to which a [symbolic reference](#) applies (possibly chosen from several [homonymous](#) alternatives) is called **reference binding**.

[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](#).

[roaming](#)

Roaming: indicates that the nodes returned by an expression could be anywhere in the tree, which inevitably means that the construct cannot be evaluated using streaming.

same key

Within a map, no two entries have the **same key**. Two atomic values K1 and K2 are the **same key** for this purpose if the relation `op:same-key(K1, K2, $UCC)` holds.

scanning expression

A `RelativePathExpr` is a **scanning expression** if and only if it is syntactically equivalent to some motionless pattern.

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 the mandatory requirements of this specification connected with the `xsl:import-schema` declaration, the `[xsl:]validation` and `[xsl:]type` attributes, and the ability to handle input documents whose nodes have type annotations other than `xs:untyped` and `xs:untypedAtomic`. The mandatory requirements of this specification are taken to include the mandatory requirements of XPath 3.0, as described in [\[XPath 3.0\]](#). 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.

secondary result

Zero or more **secondary results**: each secondary result can be any sequence of items (as defined in [\[XDM 3.0\]](#)).

selection pattern

A **selection pattern** uses a subset of the syntax for path expressions, and is defined to match a node if the corresponding path expression would select the node. Selection patterns may also be formed by combining other patterns using union, intersection, and difference operators.

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, atomic values, and function items. The way that the resulting sequence is used depends on the containing instruction.

SequenceType

A **SequenceType** constrains the type and number of items in a sequence. The term is used both to denote the concept, and to refer to the syntactic form in which sequence types are expressed in the XPath grammar: specifically `SequenceTypeXP30` in [\[XPath 3.0\]](#), or `SequenceTypeXP31` in [\[XPath 3.1\]](#), depending on whether or not the [XPath 3.1 Feature](#) is implemented.

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 [principal result](#) or [secondary result](#), it is still possible that errors may occur in serializing that result. For example, it may be impossible to serialize the result 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 [26 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](#)

A **simplified stylesheet**, which is a subtree rooted at a [literal result element](#), as described in [3.8 Simplified Stylesheet Modules](#). This is first converted to a [standard stylesheet module](#) by wrapping it in an `xsl:stylesheet` element using the transformation described in [3.8 Simplified Stylesheet Modules](#).

[singleton focus](#)

A **singleton focus** based on an item J has the [context item](#) (and therefore the [context node](#), if J is a node) set to J , and the [context position](#) and [context size](#) both set to 1 (one).

[snapshot](#)

A **snapshot** of a node N is a deep copy of N , as produced by the [xsl:copy-of](#) instruction with `copy-namespaces` set to yes, `copy-accumulators` set to yes, and `validation` set to preserve, with the additional property that for every ancestor of N , the copy also has a corresponding ancestor whose name, node-kind, and base URI are the same as the corresponding ancestor of N , and that has copies of the attributes, namespaces and accumulator values of the corresponding ancestor of N . But the ancestor has a type annotation of `xs:anyType`, has the properties `nilled`, `is-id`, and `is-idref` set to false, and has no children other than the child that is a copy of N or one of its ancestors.

[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 [global context item](#) if any, documents containing nodes present in the [initial match selection](#), documents containing nodes supplied as the values of [stylesheet parameters](#), documents obtained from the results of functions such as [document](#), [doc^{FO30}](#), and [collection^{FO30}](#), documents read using the [xsl:source-document](#) instruction, 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.

standard attributes

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

standard error namespace

The **standard error namespace** <http://www.w3.org/2005/xqt-errors> is used for error codes defined in this specification and related specifications. It is also used for the names of certain predefined variables accessible within the scope of an [`xsl:catch`](#) element.

standard function namespace

The **standard function namespace** <http://www.w3.org/2005/xpath-functions> is used for functions in the function library defined in [\[Functions and Operators 3.0\]](#) and for standard functions defined in this specification.

standard stylesheet module

A **standard stylesheet module**, which is a subtree rooted at an [`xsl:stylesheet`](#) or [`xsl:transform`](#) element.

static error

An error that can be 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**.

static expression

A **static expression** is an XPath [expression](#) whose value must be computed during static analysis of the stylesheet.

static parameter

A [static variable](#) declared using an [`xsl:param`](#) element is referred to as a **static parameter**.

static type

The **static type** of a [construct](#) is such that all values produced by evaluating the construct will conform to that type. The static type of a construct is a [U-type](#).

static variable

A [top-level variable-binding element](#) having the attribute `static="yes"` declares a **static variable**: that is, a [global variable](#) whose value is known during static analysis of the stylesheet.

streamability category

Stylesheet functions belong to one of a number of **streamability categories**: the choice of category characterizes the way in which the function handles streamed input.

streamable mode

A **streamable mode** is a [mode](#) that is declared in an [`xsl:mode`](#) declaration with the attribute `streamable="yes"`.

streamed document

A **streamed document** is a [source tree](#) that is processed using streaming, that is, without constructing a complete tree of nodes in memory.

[streamed node](#)

A **streamed node** is a node in a [streamed document](#).

[streaming](#)

The term **streaming** refers to a manner of processing in which XML documents (such as source and result documents) are not represented by a complete tree of nodes occupying memory proportional to document size, but instead are processed “on the fly” as a sequence of events, similar in concept to the stream of events notified by an XML parser to represent markup in lexical XML.

[streaming feature](#)

A processor that claims conformance with the **streaming feature** MUST use streamed processing in cases where (a) streaming is requested (for example by using the attribute `streamable="yes"` on [xsl:mode](#), or on the [xsl:source-document](#) instruction) and (b) the constructs in question are [guaranteed-streamable](#) according to this specification.

[streaming parameter](#)

The first [parameter](#) of a [declared-streamable stylesheet function](#) is referred to as a **streaming parameter**.

[striding](#)

Striding: indicates that the result of a construct contains a sequence of streamed nodes, in document order, that are peers in the sense that none of them is an ancestor or descendant of any other.

[string value](#)

The term **string value** is defined in [Section 5.13 string-value Accessor](#)^{DM30}. 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 **stylesheet** consists of one or more packages: specifically, one [top-level package](#) and zero or more [library packages](#).

[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](#) (subject to visibility rules).

[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 [package](#) 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.

sweep

Every construct has a **sweep**, which is a measure of the extent to which the current position in the input stream moves during the evaluation of the expression. The sweep is one of: [motionless](#), [consuming](#), or [free-ranging](#).

symbolic identifier

The **symbolic identifier** of a [component](#) is a composite name used to identify the component uniquely within a package. The symbolic identifier comprises the kind of component (stylesheet function, named template, accumulator, attribute set, global variable, key, or mode), the [expanded QName](#) of the component (namespace URI plus local name), and in the case of stylesheet functions, the [arity](#).

symbolic reference

The [declaration](#) of a component includes constructs that can be interpreted as references to other [components](#) by means of their [symbolic identifiers](#). These constructs are generically referred to as **symbolic references**. Examples of constructs that give rise to symbolic references are the name attribute of [xsl:call-template](#); the [xsl:]use-attribute-sets attribute of [xsl:copy](#), [xsl:element](#), and [literal result elements](#); the explicit or implicit mode attribute of [xsl:apply-templates](#); XPath variable references referring to global variables; XPath static function calls (including partial function applications) referring to [stylesheet functions](#); and named function references (example: my:f#1) referring to stylesheet functions.

tail position

An [instruction](#) J is in a **tail position** within a [sequence constructor](#) SC if it satisfies one of the following conditions:

- J is the last instruction in SC , ignoring any [xsl:fallback](#) instructions.
- J is in a [tail position](#) within the sequence constructor that forms the body of an [xsl:if](#) instruction that is itself in a [tail position](#) within SC .
- J is in a [tail position](#) within the sequence constructor that forms the body of an [xsl:when](#) or [xsl:otherwise](#) branch of an [xsl:choose](#) instruction that is itself in a [tail position](#) within SC .
- J is in a [tail position](#) within the sequence constructor that forms the body of an [xsl:try](#) instruction that is itself in a [tail position](#) within SC (that is, it is immediately followed by an [xsl:catch](#) element, ignoring any [xsl:fallback](#) elements).
- J is in a [tail position](#) within the sequence constructor that forms the body of an [xsl:catch](#) element within an [xsl:try](#) instruction that is itself in a [tail position](#) within SC .

target expression

The string that results from evaluating the expression in the [xpath](#) attribute is referred to as the **target expression**.

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](#); this sequence constructor is evaluated to determine the result of the template. A template can serve either as a [template rule](#), invoked by matching items 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 selected items (often but not necessarily 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](#).

text value template

In a text node that is designated as a **text value template**, [expressions](#) can be used by surrounding each expression with curly brackets ({}).

top-level

An element occurring as a child of an [xsl:package](#), [xsl:stylesheet](#), [xsl:transform](#), or [xsl:override](#) element is called a **top-level** element.

top-level package

For a given transformation, one [package](#) functions as the **top-level package**. The complete [stylesheet](#) is assembled by finding the packages referenced directly or indirectly from the top-level package using [xsl:use-package](#) declarations: see [3.5.2 Dependencies between Packages](#).

transmission

An operand usage of **transmission** indicates that the construct will (potentially) return a supplied node as part of its result to the calling construct (that is, to its parent in the construct tree).

traversal

A **traversal** of a tree is a sequence of [traversal events](#).

traversal-event

A **traversal event** (shortened to **event** in this section) is a pair comprising a phase (start or end) and a node.

tree

The term **tree** is used (as in [\[XDM 3.0\]](#)) to refer to the aggregate consisting of a parentless node together with all its descendant nodes, plus all their attributes and namespaces.

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](#)^{DM30}.

type error

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.

type-adjusted posture and sweep

The **type-adjusted posture and sweep** of a construct C , with respect to a type T , are the posture and sweep established by applying the general streamability rules to a construct D whose single operand is the construct C , where the operand usage of C in D is the type-determined usage based on the required type T .

type-determined usage

The **type-determined usage** of an operand is as follows: if the required type (ignoring occurrence indicator) is `function(*)` or a subtype thereof, then inspection; if the required type (ignoring occurrence indicator) is an atomic or union type, then absorption; otherwise navigation.

typed value

The term **typed value** is defined in [Section 5.15 typed-value Accessor](#)^{DM30}. Every node, other than an element whose type annotation identifies it as having 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.

U-type

A **U-type** is a set of fundamental item types.

unnamed mode

The **unnamed mode** is the default mode used when no `mode` attribute is specified on an xsl:apply-templates instruction or xsl:template declaration, unless a different default mode has been specified using the `[xsl:]default-mode` attribute of a containing element.

URI Reference

Within this specification, the term **URI Reference**, unless otherwise stated, refers to a string in the lexical space of the `xs:anyURI` datatype as defined in [\[XML Schema Part 2\]](#).

use

If a package Q contains an xsl:use-package element that references package P , then package Q is said to **use** package P . In this relationship package Q is referred to as the **using** package, package P as the **used** package.

user-defined data element

In addition to declarations, the xsl:stylesheet element may contain among its children 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**.

vacuous

An item is **vacuous** if it is one of the following: a zero-length text node; a document node with no children; an atomic value which, on casting to `xs:string`, produces a zero-length string; or (when XPath 3.1 is supported) an array which on flattening using the array:flatten^{FO31} function produces either an empty sequence or a sequence consisting entirely of vacuous items.

value

A variable is a binding between a name and a value. The **value** of a variable is any sequence (of nodes, atomic values, and/or function items), as defined in [\[XDM 3.0\]](#).

value template

Collectively, attribute value templates and text value templates are referred to as **value templates**.

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**.

visibility

The **visibility** of a [component](#) is one of: `private`, `public`, `abstract`, `final`, or `hidden`.

whitespace text node

A **whitespace text node** is a text node whose content consists entirely of whitespace characters (that is, `#x09`, `#xA`, `#xD`, or `#x20`).

XML namespace

The **XML namespace**, defined in [Namespaces in XML](#) 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](#)^{XP30}. 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 3.0 and XPath 1.0.

XPath 3.1 Feature

A processor that claims conformance with the **XPath 3.1 feature** MUST implement XPath 3.1 (including [\[XPath 3.1\]](#), [\[XDM 3.1\]](#), [\[XSLT and XQuery Serialization 3.1\]](#), and [\[Functions and Operators 3.1\]](#)).

XSLT 1.0 behavior

An element in the stylesheet is processed with **XSLT 1.0 behavior** if its [effective version](#) is equal to 1.0.

XSLT 1.0 compatibility feature

A processor that claims conformance with the **XSLT 1.0 compatibility feature** MUST support the processing of stylesheet instructions and XPath expressions with [XSLT 1.0 behavior](#), as defined in [3.9 Backwards Compatible Processing](#).

XSLT 2.0 behavior

An element is processed with **XSLT 2.0 behavior** if its [effective version](#) is equal to 2.0.

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.

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:accept

Syntax summary for element xsl:accept

Model:

```
<xsl:accept
    component = "template" | "function" | "attribute-set" | "variable" |
    "mode" | "*"
    names = tokens
    visibility = "public" | "private" | "final" | "abstract" | "hidden" />
```

Permitted parent elements:

- [xsl:use-package](#)

xsl:accumulator

Syntax summary for element xsl:accumulator

Category: declaration

Model:

```
<xsl:accumulator
    name = eqname
    initial-value = expression
    as? = sequence-type
    streamable? = boolean >
    <!-- Content: xsl:accumulator-rule+ -->
</xsl:accumulator>
```

Permitted parent elements:

- [xsl:package](#)
- [xsl:stylesheet](#)
- [xsl:transform](#)

xsl:accumulator-rule

Syntax summary for element xsl:accumulator-rule

Model:

```
<xsl:accumulator-rule
    match = pattern
```

```

phase? = "start" | "end"
select? = expression >
<!-- Content: sequence-constructor -->
</xsl:accumulator-rule>
```

Permitted parent elements:

- [xsl:accumulator](#)

[xsl:analyze-string](#)

Syntax summary for element 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](#)

Syntax summary for 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](#)

Syntax summary for 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:assert

Syntax summary for element xsl:assert

Category: instruction

Model:

```
<xsl:assert
  test = expression
  select? = expression
  error-code? = { eqname } >
  <!-- Content: sequence-constructor -->
</xsl:assert>
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

xsl:attribute

Syntax summary for element xsl:attribute

Category: instruction

Model:

```
<xsl:attribute
  name = { qname }
  namespace? = { uri }
  select? = expression
  separator? = { string }
  type? = eqname
  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](#)

Syntax summary for element xsl:attribute-set

Category: declaration

Model:

```
<xsl:attribute-set
  name = eqname
  use-attribute-sets? = eqnames
  visibility? = "public" | "private" | "final" | "abstract"
  streamable? = boolean >
  <!-- Content: xsl:attribute* -->
</xsl:attribute-set>
```

Permitted parent elements:

- [xsl:package](#)
- [xsl:stylesheet](#)
- [xsl:transform](#)
- [xsl:override](#)

[xsl:break](#)

Syntax summary for element xsl:break

Category: instruction

Model:

```
<xsl:break
  select? = expression >
  <!-- Content: sequence-constructor -->
</xsl:break>
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

xsl:call-template

Syntax summary for element xsl:call-template

Category: instruction

Model:

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

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

xsl:catch

Syntax summary for element xsl:catch

Model:

```
<xsl:catch
  errors? = tokens
  select? = expression >
  <!-- Content: sequence-constructor -->
</xsl:catch>
```

Permitted parent elements:

- [xsl:try](#)

xsl:character-map

Syntax summary for element xsl:character-map

Category: declaration

Model:

```
<xsl:character-map
  name = eqname
  use-character-maps? = eqnames >
  <!-- Content: (xsl:output-character*) -->
</xsl:character-map>
```

Permitted parent elements:

- [xsl:package](#)
- [xsl:stylesheet](#)

- [xsl:transform](#)

[xsl:choose](#)

Syntax summary for element 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](#)

Syntax summary for 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:context-item](#)

Syntax summary for element xsl:context-item

Model:

```
<xsl:context-item
  as? = item-type
  use? = "required" | "optional" | "absent" />
```

Permitted parent elements:

- [xsl:template](#)

xsl:copy

Syntax summary for element *xsl:copy*

Category: instruction

Model:

```
<xsl:copy
  select? = expression
  copy-namespaces? = boolean
  inherit-namespaces? = boolean
  use-attribute-sets? = eqnames
  type? = eqname
  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

Syntax summary for element *xsl:copy-of*

Category: instruction

Model:

```
<xsl:copy-of
  select = expression
  copy-accumulators? = boolean
  copy-namespaces? = boolean
  type? = eqname
  validation? = "strict" | "lax" | "preserve" | "strip" />
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

xsl:decimal-format

Syntax summary for element *xsl:decimal-format*

Category: declaration

Model:

```
<xsl:decimal-format
    name? = eqname
    decimal-separator? = char
    grouping-separator? = char
    infinity? = string
    minus-sign? = char
    exponent-separator? = char
    NaN? = string
    percent? = char
    per-mille? = char
    zero-digit? = char
    digit? = char
    pattern-separator? = char />
```

Permitted parent elements:

- [xsl:package](#)
- [xsl:stylesheet](#)
- [xsl:transform](#)

[xsl:document](#)

Syntax summary for element xsl:document

Category: instruction

Model:

```
<xsl:document
    validation? = "strict" | "lax" | "preserve" | "strip"
    type? = eqname >
    <!-- Content: sequence-constructor -->
</xsl:document>
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

[xsl:element](#)

Syntax summary for element xsl:element

Category: instruction

Model:

```
<xsl:element
    name = { qname }
    namespace? = { uri }>
```

```

inherit-namespaces? = boolean
use-attribute-sets? = eqnames
type? = eqname
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:evaluate

Syntax summary for element xsl:evaluate

Category: instruction

Model:

```

<xsl:evaluate
  xpath = expression
  as? = sequence-type
  base-uri? = { uri }
  with-params? = expression
  context-item? = expression
  namespace-context? = expression
  schema-aware? = { boolean } >
  <!-- Content: (xsl:with-param | xsl:fallback)* -->
</xsl:evaluate>
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

xsl:expose

Syntax summary for element xsl:expose

Model:

```

<xsl:expose
  component = "template" | "function" | "attribute-set" | "variable" |
  "mode" | "*"
  names = tokens
  visibility = "public" | "private" | "final" | "abstract" />
```

Permitted parent elements:

- [xsl:package](#)

xsl:fallback

Syntax summary for element xsl:fallback

Category: instruction

Model:

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

Permitted parent elements:

- [xsl:analyze-string](#)
- [xsl:evaluate](#)
- [xsl:fork](#)
- [xsl:merge](#)
- [xsl:next-match](#)
- [xsl:try](#)
- any XSLT element whose content model is *sequence constructor*
- any literal result element

xsl:for-each

Syntax summary for 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

Syntax summary for 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
  composite? = boolean
  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:fork](#)

Syntax summary for element xsl:fork

Category: instruction

Model:

```

<xsl:fork>
  <!-- Content: (xsl:fallback*, ((xsl:sequence, xsl:fallback*)* | (xsl:for-
each-group, xsl:fallback*)) -->
</xsl:fork>
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

[xsl:function](#)

Syntax summary for element xsl:function

Category: declaration

Model:

```

<xsl:function
  name = eqname
  as? = sequence-type
  visibility? = "public" | "private" | "final" | "abstract"
  streamability? = "unclassified" | "absorbing" | "inspection" | "filter" |
  "shallow-descent" | "deep-descent" | "ascent" | eqname
  override-extension-function? = boolean
  [override]? = boolean
```

```

new-each-time? = "yes" | "true" | "1" | "no" | "false" | "0" | "maybe"
cache? = boolean >
<!-- Content: (xsl:param* , sequence-constructor) -->
</xsl:function>
```

Permitted parent elements:

- [xsl:package](#)
- [xsl:stylesheet](#)
- [xsl:transform](#)
- [xsl:override](#)

[xsl:global-context-item](#)

Syntax summary for element xsl:global-context-item

Category: declaration

Model:

```
<xsl:global-context-item
  as? = item-type
  use? = "required" | "optional" | "absent" />
```

Permitted parent elements:

- [xsl:package](#)
- [xsl:stylesheet](#)
- [xsl:transform](#)

[xsl:if](#)

Syntax summary for element 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](#)

*Syntax summary for element xsl:import**Category:* declaration*Model:*

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

Permitted parent elements:

- [xsl:package](#)
- [xsl:stylesheet](#)
- [xsl:transform](#)

xsl:import-schema*Syntax summary for element xsl:import-schema**Category:* declaration*Model:*

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

Permitted parent elements:

- [xsl:package](#)
- [xsl:stylesheet](#)
- [xsl:transform](#)

xsl:include*Syntax summary for element xsl:include**Category:* declaration*Model:*

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

Permitted parent elements:

- [xsl:package](#)
- [xsl:stylesheet](#)
- [xsl:transform](#)

xsl:iterate

Syntax summary for element xsl:iterate

Category: instruction

Model:

```
<xsl:iterate
  select = expression >
  <!-- Content: (xsl:param*, xsl:on-completion??, sequence-constructor) -->
</xsl:iterate>
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

xsl:key

Syntax summary for element xsl:key

Category: declaration

Model:

```
<xsl:key
  name = eqname
  match = pattern
  use? = expression
  composite? = boolean
  collation? = uri >
  <!-- Content: sequence-constructor -->
</xsl:key>
```

Permitted parent elements:

- [xsl:package](#)
- [xsl:stylesheet](#)
- [xsl:transform](#)

xsl:map

Syntax summary for element xsl:map

Category: instruction

Model:

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

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

xsl:map-entry

Syntax summary for element xsl:map-entry

Category: instruction

Model:

```
<xsl:map-entry>
  key = expression
  select? = expression >
  <!-- Content: sequence-constructor -->
</xsl:map-entry>
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

xsl:matching-substring

Syntax summary for element xsl:matching-substring

Model:

```
<xsl:matching-substring>
  <!-- Content: sequence-constructor -->
</xsl:matching-substring>
```

Permitted parent elements:

- [xsl:analyze-string](#)

xsl:merge

Syntax summary for element xsl:merge

Category: instruction

Model:

```
<xsl:merge>
  <!-- Content: (xsl:merge-source+, xsl:merge-action, xsl:fallback\*) -->
</xsl:merge>
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

[xsl:merge-action](#)

Syntax summary for element xsl:merge-action

Model:

```
<xsl:merge-action>
  <!-- Content: sequence-constructor -->
</xsl:merge-action>
```

Permitted parent elements:

- [xsl:merge](#)

[xsl:merge-key](#)

Syntax summary for element xsl:merge-key

Model:

```
<xsl:merge-key
  select? = expression
  lang? = { language }
  order? = { "ascending" | "descending" }
  collation? = { uri }
  case-order? = { "upper-first" | "lower-first" }
  data-type? = { "text" | "number" | eqname } >
  <!-- Content: sequence-constructor -->
</xsl:merge-key>
```

Permitted parent elements:

- [xsl:merge-source](#)

[xsl:merge-source](#)

Syntax summary for element xsl:merge-source

Model:

```
<xsl:merge-source
  name? = ncname
```

```

for-each-item? = expression
for-each-source? = expression
select = expression
streamable? = boolean
use-accumulators? = tokens
sort-before-merge? = boolean
validation? = "strict" | "lax" | "preserve" | "strip"
type? = eqname >
<!-- Content: xsl:merge-key+ -->
</xsl:merge-source>
```

Permitted parent elements:

- [xsl:merge](#)

[xsl:message](#)

Syntax summary for element xsl:message

Category: instruction

Model:

```

<xsl:message
  select? = expression
  terminate? = { boolean }
  error-code? = { eqname } >
  <!-- Content: sequence-constructor -->
</xsl:message>
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

[xsl:mode](#)

Syntax summary for element xsl:mode

Category: declaration

Model:

```

<xsl:mode
  name? = eqname
  streamable? = boolean
  use-accumulators? = tokens
  on-no-match? = "deep-copy" | "shallow-copy" | "deep-skip" | "shallow-skip"
  | "text-only-copy" | "fail"
  on-multiple-match? = "use-last" | "fail"
  warning-on-no-match? = boolean
```

```

warning-on-multiple-match? = boolean
typed? = boolean | "strict" | "lax" | "unspecified"
visibility? = "public" | "private" | "final" />

```

Permitted parent elements:

- [xsl:package](#)
- [xsl:stylesheet](#)
- [xsl:transform](#)

[xsl:namespace](#)

Syntax summary for element xsl:namespace

Category: instruction

Model:

```

<xsl:namespace
  name = { ncname }
  select? = expression >
  <!-- Content: sequence-constructor -->
/>
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

[xsl:namespace-alias](#)

Syntax summary for element xsl:namespace-alias

Category: declaration

Model:

```

<xsl:namespace-alias
  stylesheet-prefix = prefix | "#default"
  result-prefix = prefix | "#default" />
```

Permitted parent elements:

- [xsl:package](#)
- [xsl:stylesheet](#)
- [xsl:transform](#)

[xsl:next-iteration](#)

Syntax summary for element xsl:next-iteration

Category: instruction

Model:

```
<xsl:next-iteration>
  <!-- Content: (xsl:with-param*) -->
</xsl:next-iteration>
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

[**xsl:next-match**](#)

Syntax summary for element 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**](#)

Syntax summary for 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**](#)

Syntax summary for element xsl:number

Category: instruction

Model:

```
<xsl:number
  value? = expression
  select? = expression
  level? = "single" | "multiple" | "any"
  count? = pattern
  from? = pattern
  format? = { string }
  lang? = { language }
  letter-value? = { "alphabetic" | "traditional" }
  ordinal? = { string }
  start-at? = { string }
  grouping-separator? = { char }
  grouping-size? = { integer } />
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

xsl:on-completion

Syntax summary for element xsl:on-completion

Model:

```
<xsl:on-completion
  select? = expression >
  <!-- Content: sequence-constructor -->
</xsl:on-completion>
```

Permitted parent elements:

- [xsl:iterate](#)

xsl:on-empty

Syntax summary for element xsl:on-empty

Category: instruction

Model:

```
<xsl:on-empty
  select? = expression >
  <!-- Content: sequence-constructor -->
</xsl:on-empty>
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

xsl:on-non-empty

Syntax summary for element xsl:on-non-empty

Category: instruction

Model:

```
<xsl:on-non-empty
  select? = expression >
  <!-- Content: sequence-constructor -->
</xsl:on-non-empty>
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

xsl:otherwise

Syntax summary for element xsl:otherwise

Model:

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

Permitted parent elements:

- [xsl:choose](#)

xsl:output

Syntax summary for element xsl:output

Category: declaration

Model:

```
<xsl:output
  name? = eqname
  method? = "xml" | "html" | "xhtml" | "text" | "json" | "adaptive" | eqname
  allow-duplicate-names? = boolean
  build-tree? = boolean
  byte-order-mark? = boolean
  cdata-section-elements? = eqnames
  doctype-public? = string
  doctype-system? = string
  encoding? = string
```

```

escape-uri-attributes? = boolean
html-version? = decimal
include-content-type? = boolean
indent? = boolean
item-separator? = string
json-node-output-method? = "xml" | "html" | "xhtml" | "text" | eqname
media-type? = string
normalization-form? = "NFC" | "NFD" | "NFKC" | "NFKD" | "fully-normalized"
| "none" | nmtoken
omit-xml-declaration? = boolean
parameter-document? = uri
standalone? = boolean | "omit"
suppress-indentation? = eqnames
undeclare-prefixes? = boolean
use-character-maps? = eqnames
version? = nmtoken />

```

Permitted parent elements:

- [xsl:package](#)
- [xsl:stylesheet](#)
- [xsl:transform](#)

xsl:output-character

Syntax summary for element xsl:output-character

Model:

```
<xsl:output-character
  character = char
  string = string />
```

Permitted parent elements:

- [xsl:character-map](#)

xsl:override

Syntax summary for element xsl:override

Model:

```
<xsl:override>
  <!-- Content: (xsl:template | xsl:function | xsl:variable | xsl:param |
    xsl:attribute-set)* -->
</xsl:override>
```

Permitted parent elements:

- [xsl:use-package](#)

[xsl:package](#)

Syntax summary for element xsl:package

Model:

```
<xsl:package
    id? = id
    name? = uri
    package-version? = string
    version = decimal
    input-type-annotations? = "preserve" | "strip" | "unspecified"
    declared-modes? = boolean
    default-mode? = eqname | "#unnamed"
    default-validation? = "preserve" | "strip"
    default-collation? = uris
    extension-element-prefixes? = prefixes
    exclude-result-prefixes? = prefixes
    expand-text? = boolean
    use-when? = expression
    xpath-default-namespace? = uri >
    <!-- Content: ((xsl:expose | declarations)*)
</xsl:package>
```

Permitted parent elements:

- None

[xsl:param](#)

Syntax summary for element xsl:param

Category: declaration

Model:

```
<xsl:param
    name = eqname
    select? = expression
    as? = sequence-type
    required? = boolean
    tunnel? = boolean
    static? = boolean >
    <!-- Content: sequence-constructor -->
</xsl:param>
```

Permitted parent elements:

- [xsl:package](#)

- [xsl:stylesheet](#)
- [xsl:transform](#)
- [xsl:override](#)
- [xsl:function](#)
- [xsl:template](#)
- [xsl:iterate](#)

[xsl:perform-sort](#)

Syntax summary for element 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](#)

Syntax summary for element xsl:preserve-space

Category: declaration

Model:

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

Permitted parent elements:

- [xsl:package](#)
- [xsl:stylesheet](#)
- [xsl:transform](#)

[xsl:processing-instruction](#)

Syntax summary for element 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

Syntax summary for element xsl:result-document

Category: instruction

Model:

```
<xsl:result-document
  format? = { eqname }
  href? = { uri }
  validation? = "strict" | "lax" | "preserve" | "strip"
  type? = eqname
  method? = { "xml" | "html" | "xhtml" | "text" | "json" | "adaptive" |
    eqname }
  allow-duplicate-names? = { boolean }
  build-tree? = { boolean }
  byte-order-mark? = { boolean }
  cdata-section-elements? = { eqnames }
  doctype-public? = { string }
  doctype-system? = { string }
  encoding? = { string }
  escape-uri-attributes? = { boolean }
  html-version? = { decimal }
  include-content-type? = { boolean }
  indent? = { boolean }
  item-separator? = { string }
  json-node-output-method? = { "xml" | "html" | "xhtml" | "text" | eqname }
  media-type? = { string }
  normalization-form? = { "NFC" | "NFD" | "NFKC" | "NFKD" | "fully-
    normalized" | "none" | nmtoken }
  omit-xml-declaration? = { boolean }
  parameter-document? = { uri }
  standalone? = { boolean | "omit" }
  suppress-indentation? = { eqnames }
  undeclare-prefixes? = { boolean }
  use-character-maps? = eqnames
  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

Syntax summary for element xsl:sequence

Category: instruction

Model:

```
<xsl:sequence
    select? = expression >
    <!-- Content: sequence-constructor -->
</xsl:sequence>
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

xsl:sort

Syntax summary for element xsl:sort

Model:

```
<xsl:sort
    select? = expression
    lang? = { language }
    order? = { "ascending" | "descending" }
    collation? = { uri }
    stable? = { boolean }
    case-order? = { "upper-first" | "lower-first" }
    data-type? = { "text" | "number" | eqname } >
    <!-- Content: sequence-constructor -->
</xsl:sort>
```

Permitted parent elements:

- [xsl:apply-templates](#)
- [xsl:for-each](#)
- [xsl:for-each-group](#)
- [xsl:perform-sort](#)

xsl:source-document

Syntax summary for element *xsl:source-document*

Category: instruction

Model:

```
<xsl:source-document
  href = { uri }
  streamable? = boolean
  use-accumulators? = tokens
  validation? = "strict" | "lax" | "preserve" | "strip"
  type? = eqname >
  <!-- Content: sequence-constructor -->
</xsl:source-document>
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

xsl:strip-space

Syntax summary for element *xsl:strip-space*

Category: declaration

Model:

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

Permitted parent elements:

- [xsl:package](#)
- [xsl:stylesheet](#)
- [xsl:transform](#)

xsl:stylesheet

Syntax summary for element *xsl:stylesheet*

Model:

```
<xsl:stylesheet
  id? = id
  version = decimal
  default-mode? = eqname | "#unnamed"
  default-validation? = "preserve" | "strip"
  input-type-annotations? = "preserve" | "strip" | "unspecified"
```

```

default-collation? = uri
extension-element-prefixes? = prefixes
exclude-result-prefixes? = prefixes
expand-text? = boolean
use-when? = expression
xpath-default-namespace? = uri >
<!-- Content: (declarations) -->
</xsl:stylesheet>
```

xsl:template

Syntax summary for element xsl:template

Category: declaration

Model:

```

<xsl:template
  match? = pattern
  name? = eqname
  priority? = decimal
  mode? = tokens
  as? = sequence-type
  visibility? = "public" | "private" | "final" | "abstract" >
  <!-- Content: (xsl:context-item?, xsl:param*, sequence-constructor) -->
</xsl:template>
```

Permitted parent elements:

- [xsl:package](#)
- [xsl:stylesheet](#)
- [xsl:transform](#)
- [xsl:override](#)

xsl:text

Syntax summary for element xsl:text

Category: instruction

Model:

```

<xsl:text
  [disable-output-escaping]? = boolean >
  <!-- Content: #PCDATA -->
</xsl:text>
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*

- any literal result element

xsl:transform

Syntax summary for element xsl:transform

Model:

```
<xsl:transform
  id? = id
  version = decimal
  default-mode? = eqname | "#unnamed"
  default-validation? = "preserve" | "strip"
  input-type-annotations? = "preserve" | "strip" | "unspecified"
  default-collation? = uris
  extension-element-prefixes? = prefixes
  exclude-result-prefixes? = prefixes
  expand-text? = boolean
  use-when? = expression
  xpath-default-namespace? = uri >
  <!-- Content: (declarations) -->
</xsl:transform>
```

xsl:try

Syntax summary for element xsl:try

Category: instruction

Model:

```
<xsl:try
  select? = expression
  rollback-output? = boolean >
  <!-- Content: (sequence-constructor, xsl:catch, (xsl:catch |
xsl:fallback)*) -->
</xsl:try>
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

xsl:use-package

Syntax summary for element xsl:use-package

Category: declaration

Model:

```
<xsl:use-package
  name = uri
  package-version? = string >
  <!-- Content: (xsl:accept | xsl:override)* -->
</xsl:use-package>
```

Permitted parent elements:

- [xsl:package](#)
- [xsl:stylesheet](#)
- [xsl:transform](#)

[xsl:value-of](#)

Syntax summary for element xsl:value-of

Category: instruction

Model:

```
<xsl:value-of
  select? = expression
  separator? = { string }
  [disable-output-escaping]? = boolean >
  <!-- Content: sequence-constructor -->
</xsl:value-of>
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

[xsl:variable](#)

Syntax summary for element xsl:variable

Category: declaration instruction

Model:

```
<xsl:variable
  name = eqname
  select? = expression
  as? = sequence-type
  static? = boolean
  visibility? = "public" | "private" | "final" | "abstract" >
  <!-- Content: sequence-constructor -->
</xsl:variable>
```

Permitted parent elements:

- [xsl:package](#)
- [xsl:stylesheet](#)
- [xsl:transform](#)
- [xsl:override](#)
- [xsl:function](#)
- any XSLT element whose content model is *sequence constructor*
- any literal result element

[xsl:when](#)

Syntax summary for element xsl:when

Model:

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

Permitted parent elements:

- [xsl:choose](#)

[xsl:where-populated](#)

Syntax summary for element xsl:where-populated

Category: instruction

Model:

```
<xsl:where-populated>
  <!-- Content: sequence-constructor -->
</xsl:where-populated>
```

Permitted parent elements:

- any XSLT element whose content model is *sequence constructor*
- any literal result element

[xsl:with-param](#)

Syntax summary for element xsl:with-param

Model:

```
<xsl:with-param
  name = eqname
  select? = expression
```

```

as? = sequence-type
tunnel? = boolean >
<!-- Content: sequence-constructor -->
</xsl:with-param>

```

Permitted parent elements:

- [xsl:apply-templates](#)
- [xsl:apply-imports](#)
- [xsl:call-template](#)
- [xsl:evaluate](#)
- [xsl:next-match](#)
- [xsl:next-iteration](#)

E Summary of Error Conditions (Non-Normative)

This appendix provides a summary of error conditions that a processor may signal. This list includes all error codes defined in this specification, but this is not an exhaustive list of all errors that can occur. Implementations **MUST** signal errors using these error codes, and applications can test for these codes; however, when more than one rule in the specification is violated, different processors will not necessarily signal the same error code. Implementations are not **REQUIRED** to signal errors using the descriptive text used here.

Note:

The appendix is non-normative because the same information is given normatively elsewhere.

Static errors

[ERR XTSE0010](#)

It is a [static error](#) 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](#), an [accumulator](#), or a [character map](#); except that the name `xsl:initial-template` is permitted as a template name.

[ERR XTSE0085](#)

It is a [static error](#) to use a [reserved namespace](#) in the name of any [extension function](#) or [extension instruction](#), other than a function or instruction defined in this specification or in a normatively referenced specification. It is a [static error](#) to use a prefix bound to a reserved namespace in the `[xsl:]extension-element-prefixes` attribute.

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 valid instance of the type `xs:decimal` as defined in [\[XML Schema Part 2\]](#).

ERR XTSE0120

An `xsl:stylesheet`, `xsl:transform`, or `xsl:package` 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 an `xsl:stylesheet`, `xsl:transform`, or `xsl:package` 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.

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 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 a [package](#) that specifies [input-type-annotations="strip"](#) and another [stylesheet module](#) that specifies [input-type-annotations="preserve"](#), or if a stylesheet module specifies the value `strip` or `preserve` and the same value is not specified on the [xsl:package](#) element of the containing package.

ERR XTSE0270

It is a [static error](#) if within any [package](#) the same [NameTest^{XP30}](#) appears in both an [xsl:strip-space](#) and an [xsl:preserve-space](#) declaration if both have the same [import precedence](#). Two NameTests are considered the same if they match the same set of names (which can be determined by comparing them after expanding namespace prefixes to URIs).

ERR XTSE0280

In the case of a prefixed [lexical QName](#) used as the value (or as part of 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 [lexical 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 [Pattern30](#).

ERR XTSE0350

It is a [static error](#) if an unescaped left curly bracket appears in a fixed part of a 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 a 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. An [xsl:template](#) element that has no `name` attribute **MUST** have no `visibility` attribute.

ERR XTSE0530

The value of 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 XTSE0545

It is a [static error](#) if for any named or unnamed [mode](#), a package explicitly specifies two conflicting values for the same attribute in different [xsl:mode](#) declarations having the same [import precedence](#), unless there is another definition of the same attribute with higher import precedence. The attributes in question are the attributes other than `name` on the [xsl:mode](#) element.

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 the values of the `name` attribute of two sibling [xsl:param](#) elements represent the same [expanded QName](#).

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 [package](#) contains more than one non-hidden 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 [package](#) contains an [`xsl:call-template`](#) instruction whose `name` attribute does not match the `name` attribute of any [named template](#) visible in the containing [package](#) (this includes any template defined in this package, as well as templates accepted from used packages whose visibility in this package is not [hidden](#)). For more details of the process of binding the called template, see [3.5.3.5 Binding References to Components](#).

ERR XTSE0660

It is a [static error](#) if a [package](#) contains more than one non-hidden [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 two or more sibling [`xsl:with-param`](#) elements have `name` attributes that represent the same [expanded QName](#).

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 non-tunnel [template parameter](#) named `x`, unless the [`xsl:call-template`](#) instruction is processed with [XSLT 1.0 behavior](#).

ERR XTSE0690

It is a [static error](#) if a [package](#) contains both (a) a named template named `T` that is not overridden by another named template of higher import precedence and that has an [explicitly mandatory](#) non-tunnel parameter named `P`, and (b) an [`xsl:call-template`](#) instruction whose `name` attribute equals `T` and that has no non-tunnel [`xsl:with-param`](#) child element whose `name` attribute equals `P`. (All names are compared as QNames.)

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 [EQNames](#), or if it contains an EQName that does not match the `name` attribute of any [`xsl:attribute-set`](#) declaration in the containing [package](#).

ERR XTSE0730

If an [`xsl:attribute`](#) set element specifies `streamable="yes"` then every attribute set referenced in its `use-attribute-sets` attribute (if present) must also specify `streamable="yes"`.

ERR XTSE0740

It is a [static error](#) if a [stylesheet function](#) has a name that is in no namespace.

ERR XTSE0760

It is a static error if an [`xsl:param`](#) child of an [`xsl:function`](#) element has either a `select` attribute or non-empty content.

ERR XTSE0770

It is a [static error](#) for a [package](#) to contain two or more [`xsl:function`](#) declarations with the same [expanded QName](#), the same [arity](#), and the same [import precedence](#), unless there is another [`xsl:function`](#) declaration 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 within a [package](#) 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.

[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 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 a [static error](#) to specify the `collation` attribute or the `composite` 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 same [`package`](#) 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 XTSE1222

It is a [static error](#) if there are several [`xsl:key`](#) declarations in a [`package`](#) with the same key name and different effective values for the `composite` attribute.

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](#), [xsl:result-document](#), [xsl:source-document](#), or [xsl:merge-source](#) elements, 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 [package](#).

[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, suppress-indentation, 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 [EQName](#). If it is a [lexical QName](#) with no 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 a [package](#) 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 containing [package](#).

[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 [non-schema-aware processor](#) MUST signal a [static error](#) if a [package](#) includes an [xsl:import-schema](#) declaration.

[ERR XTSE1660](#)

A [non-schema-aware processor](#) MUST signal a [static error](#) if a [package](#) includes an [xsl:]type attribute; or an [xsl:]validation or [xsl:]default-validation attribute with a value other than strip, preserve, or lax; or an [xsl:mode](#) element whose typed attribute is equal to yes or strict; or an as attribute whose value is a [SequenceType](#) that can only match nodes with a type annotation other than xs:untyped or xs:untypedAtomic (for example, as="element(*, xs:integer)").

ERR XTSE2200

It is a [static error](#) if the number of [`xsl:merge-key`](#) children of a [`xsl:merge-source`](#) element is not equal to the number of [`xsl:merge-key`](#) children of another [`xsl:merge-source`](#) child of the same [`xsl:merge`](#) instruction.

ERR XTSE3000

It is a [static error](#) if no package matching the package name and version specified in an [`xsl:use-package`](#) declaration can be located.

ERR XTSE3005

It is a [static error](#) if a package is dependent on itself, where package *A* is defined as being dependent on package *B* if *A* contains an [`xsl:use-package`](#) declaration that references *B*, or if *A* contains an [`xsl:use-package`](#) declaration that references a package *C* that is itself dependent on *B*.

ERR XTSE3008

It is a [static error](#) if an [`xsl:use-package`](#) declaration appears in a [stylesheet module](#) that is not in the same [stylesheet level](#) as the [principal stylesheet module](#) of the [package](#).

ERR XTSE3010

It is a [static error](#) if the explicit exposed visibility of a component is inconsistent with its declared visibility, as defined in the above table. (This error occurs only when the component declaration has an explicit [visibility](#) attribute, and the component is also listed explicitly by name in an [`xsl:expose`](#) declaration.)

ERR XTSE3020

It is a [static error](#) if a token in the [names](#) attribute of [`xsl:expose`](#), other than a wildcard, matches no component in the containing package.

ERR XTSE3022

It is a [static error](#) if the [component](#) attribute of [`xsl:expose`](#) specifies * (meaning all component kinds) and the [names](#) attribute is not a wildcard.

ERR XTSE3025

It is a [static error](#) if the effect of an [`xsl:expose`](#) declaration would be to make a component [abstract](#), unless the component is already [abstract](#) in the absence of the [`xsl:expose`](#) declaration.

ERR XTSE3030

It is a [static error](#) if a token in the [names](#) attribute of [`xsl:accept`](#), other than a wildcard, matches no component in the used package.

ERR XTSE3032

It is a [static error](#) if the [component](#) attribute of [`xsl:accept`](#) specifies * (meaning all component kinds) and the [names](#) attribute is not a wildcard.

ERR XTSE3040

It is a [static error](#) if the visibility assigned to a component by an [`xsl:accept`](#) element is incompatible with the visibility of the corresponding component in the used package, as defined by the above table, unless the token that matches the component name is a wildcard, in which case the [`xsl:accept`](#) element is treated as not matching that component.

ERR XTSE3050

It is a [static error](#) if the [`xsl:use-package`](#) elements in a [package manifest](#) cause two or more [homonymous](#) components to be accepted with a visibility other than [hidden](#).

ERR XTSE3051

It is a [static error](#) if a token in the `names` attribute of [`xsl:accept`](#), other than a wildcard, matches the symbolic name of a component declared within an [`xsl:override`](#) child of the same [`xsl:use-package`](#) element.

[ERR XTSE3055](#)

It is a [static error](#) if a component declaration appearing as a child of [`xsl:override`](#) is [homonymous](#) with any other declaration in the using package, regardless of [import precedence](#), including any other overriding declaration in the package manifest of the using package.

[ERR XTSE3058](#)

It is a [static error](#) if a component declaration appearing as a child of [`xsl:override`](#) does not match (is not [homonymous](#) with) some component in the used package.

[ERR XTSE3060](#)

It is a [static error](#) if the component referenced by an [`xsl:override`](#) declaration has [visibility](#) other than `public` or `abstract`.

[ERR XTSE3070](#)

It is a [static error](#) if the signature of an overriding component is not [compatible](#) with the signature of the component that it is overriding.

[ERR XTSE3075](#)

It is a [static error](#) to use the component reference `xsl:original` when the overridden component has `visibility="abstract"`.

[ERR XTSE3080](#)

It is a [static error](#) if a [top-level package](#) (as distinct from a [library package](#)) contains components whose `visibility` is `abstract`.

[ERR XTSE3085](#)

It is a [static error](#), when the effective value of the `declared-modes` attribute of an [`xsl:package`](#) element is `yes`, if the package contains an explicit reference to an undeclared mode, or if it implicitly uses the unnamed mode and the unnamed mode is undeclared.

[ERR XTSE3087](#)

It is a [static error](#) if more than one [`xsl:global-context-item`](#) declaration appears within a [stylesheet module](#), or if several modules within a single [`package`](#) contain inconsistent [`xsl:global-context-item`](#) declarations

[ERR XTSE3088](#)

It is a [static error](#) if the `as` attribute is present [on the [`xsl:context-item`](#) element] when `use="absent"` is specified.

[ERR XTSE3089](#)

It is a [static error](#) if the `as` attribute is present [on the [`xsl:global-context-item`](#) element] when `use="absent"` is specified.

[ERR XTSE3105](#)

It is a [static error](#) if a template rule applicable to a mode that is defined with `typed="strict"` uses a match pattern that contains a `RelativePathExprP` whose first `StepExprP` is an `AxisStepP` whose `ForwardStepP` uses an axis whose principal node kind is `Element` and whose `NodeTest` is an `EQName` that does not correspond to the name of any global element declaration in the [in-scope schema components](#).

[ERR XTSE3120](#)

It is a [static error](#) if an [xsl:break](#) or [xsl:next-iteration](#) element appears other than in a [tail position](#) within the [sequence constructor](#) forming the body of an [xsl:iterate](#) instruction.

[ERR XTSE3125](#)

It is a [static error](#) if the [select](#) attribute of [xsl:break](#) or [xsl:on-completion](#) is present and the instruction has children.

[ERR XTSE3130](#)

It is a [static error](#) if the [name](#) attribute of an [xsl:with-param](#) child of an [xsl:next-iteration](#) element does not match the [name](#) attribute of an [xsl:param](#) child of the innermost containing [xsl:iterate](#) instruction.

[ERR XTSE3140](#)

It is a [static error](#) if the [select](#) attribute of the [xsl:try](#) element is present and the element has children other than [xsl:catch](#) and [xsl:fallback](#) elements.

[ERR XTSE3150](#)

It is a [static error](#) if the [select](#) attribute of the [xsl:catch](#) element is present unless the element has empty content.

[ERR XTSE3185](#)

It is a [static error](#) if the [select](#) attribute of [xsl:sequence](#) is present and the instruction has children other than [xsl:fallback](#).

[ERR XTSE3190](#)

It is a [static error](#) if two sibling [xsl:merge-source](#) elements have the same name.

[ERR XTSE3195](#)

If the [for-each-item](#) is present then the [for-each-source](#), [use-accumulators](#), and [streamable](#) attributes must both be absent. If the [use-accumulators](#) attribute is present then the [for-each-source](#) attribute must be present. If the [for-each-source](#) attribute is present then the [for-each-item](#) attribute must be absent.

[ERR XTSE3200](#)

It is a [static error](#) if an [xsl:merge-key](#) element with a [select](#) attribute has non-empty content.

[ERR XTSE3280](#)

It is a [static error](#) if the [select](#) attribute of the [xsl:map-entry](#) element is present unless the element has no children other than [xsl:fallback](#) elements.

[ERR XTSE3300](#)

It is a [static error](#) if the list of accumulator names [in the [use-accumulators](#) attribute] contains an invalid token, contains the same token more than once, or contains the token `#all` along with any other value; or if any token (other than `#all`) is not the name of a [declared-streamable](#) accumulator visible in the containing package.

[ERR XTSE3350](#)

It is a [static error](#) for a [package](#) to contain two or more non-hidden accumulators with the same [expanded QName](#) and the same [import precedence](#), unless there is another accumulator with the same [expanded QName](#), and a higher import precedence.

[ERR XTSE3430](#)

It is a [static error](#) if a [package](#) contains a construct that is declared to be streamable but which is not [guaranteed-streamable](#), unless the user has indicated that the processor is to handle this situation by processing

the stylesheet without streaming or by making use of processor extensions to the streamability rules where available.

ERR XTSE3440

In the case of a [template rule](#) (that is, an [`xsl:template`](#) element having a `match` attribute) appearing as a child of [`xsl:override`](#), it is a [static error](#) if the list of modes in the `mode` attribute contains `#all` or `#unnamed`, or if it contains `#default` and the default mode is the [unnamed mode](#), or if the `mode` attribute is omitted when the default mode is the [unnamed mode](#).

ERR XTSE3450

It is a [static error](#) if a variable declared with `static="yes"` is inconsistent with another static variable of the same name that is declared earlier in stylesheet tree order and that has lower [import precedence](#).

ERR XTSE3460

It is a [static error](#) if an [`xsl:apply-imports`](#) element appears in a [template rule](#) declared within an [`xsl:override`](#) element. (To invoke the template rule that is being overridden, [`xsl:next-match`](#) should therefore be used.)

ERR XTSE3470

It is a [static error](#) if the [`current-merge-group`](#) function is used within a [pattern](#).

ERR XTSE3500

It is a [static error](#) if the [`current-merge-key`](#) function is used within a [pattern](#).

ERR XTSE3520

It is a static error if a parameter to [`xsl:iterate`](#) is [implicitly mandatory](#).

ERR XTSE3540

A processor that does not provide the [higher-order functions feature](#) raises a [static error](#) if any of the following XPath constructs are found in an [expression](#), [pattern](#), [SequenceType](#), or [ItemType](#): a [`TypedFunctionTest`](#)^{XP30}, a [`NamedFunctionRef`](#)^{XP30}, an [`InlineFunctionExpr`](#)^{XP30}, or an [`ArgumentPlaceholder`](#)^{XP30}

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 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 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 XTTE0945

It is a [type error](#) to use the [`xsl:copy`](#) instruction with no `select` attribute when the context item is absent.

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 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 processed with [`XSLT 1.0 behavior`](#). With XSLT 1.0 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 result of evaluating the `group-adjacent` expression is an empty sequence or a sequence containing more than one item, unless `composite="yes"` is specified.

[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 XTTE1535](#)

It is a [type error](#) if the value of the `type` attribute of an [`xsl:copy`](#) or [`xsl:copy-of`](#) instruction refers to a complex type definition and one or more of the items being copied is an attribute node.

[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 (such as ID/IDREF constraints) are not satisfied.

ERR XTTE2230

It is a [type error](#) if some item selected by a particular merge key in one input sequence is not comparable using the XPath `le` operator with some item selected by the corresponding sort key in another input sequence.

ERR XTTE3090

It is a [type error](#) if the `xsl:context-item` child of `xsl:template` specifies that a context item is required and none is supplied by the caller, that is, if the context item is absent at the point where `xsl:call-template` is evaluated.

ERR XTTE3100

It is a [type error](#) if an `xsl:apply-templates` instruction in a particular mode selects an element or attribute whose type is `xs:untyped` or `xs:untypedAtomic` when the `typed` attribute of that mode specifies the value `yes`, `strict`, or `lax`.

ERR XTTE3110

It is a [type error](#) if an `xsl:apply-templates` instruction in a particular mode selects an element or attribute whose type is anything other than `xs:untyped` or `xs:untypedAtomic` when the `typed` attribute of that mode specifies the value `no`.

ERR XTTE3165

It is a [type error](#) if the result of evaluating the expression in the `with-params` attribute of the `xsl:evaluate` instruction is anything other than a single map of type `map(xs:QName, item()*)`.

ERR XTTE3170

It is a [type error](#) if the result of evaluating the `namespace-context` attribute of the `xsl:evaluate` instruction is anything other than a single node.

ERR XTTE3180

It is a [type error](#) if the result of evaluating the `select` expression [of the `xsl:copy` element] is a sequence of more than one item.

ERR XTTE3210

If the result of evaluating the `context-item` expression [of an `xsl:evaluate` instruction] is a sequence containing more than one item, then a [type error](#) is signaled.

ERR XTTE3375

A type error occurs if the result of evaluating the sequence constructor [within an `xsl:map` instruction] is not an instance of the required type `map(*)*`.

Dynamic errors

ERR XTDE0030

It is a [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 [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](#), whose visibility is [public](#) or [final](#).

ERR XTDE0041

It is a [dynamic error](#) if the invocation of the [stylesheet](#) specifies a function name and arity that does not match the [expanded QName](#) and arity of a named [stylesheet function](#) defined in the [stylesheet](#), whose visibility is [public](#) or [final](#).

ERR XTDE0044

It is a [dynamic error](#) if the invocation of the [stylesheet](#) specifies an [initial mode](#) when no [initial match selection](#) is supplied (either explicitly, or defaulted to the [global context item](#)).

ERR XTDE0045

It is a [dynamic error](#) if the invocation of the [stylesheet](#) specifies an [initial mode](#) and the specified mode is not eligible as an initial mode (as defined above).

ERR XTDE0050

It is a [dynamic error](#) if a stylesheet declares a visible [stylesheet parameter](#) that is [explicitly](#) or [implicitly](#) mandatory, and no value for this parameter is supplied when the stylesheet is primed. A stylesheet parameter is visible if it is not masked by another global variable or parameter with the same name and higher [import precedence](#). If the parameter is a [static parameter](#) then the value [MUST](#) be supplied prior to the static analysis phase.

ERR XTDE0160

It is a [dynamic error](#) if an element has an [effective version](#) of V (with $V < 3.0$) when the implementation does not support backwards compatible behavior for XSLT version V .

ERR XTDE0290

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

ERR XTDE0410

It is a [dynamic error](#) if the 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 [dynamic error](#) if the sequence used to construct the content of a document node contains a namespace node or attribute node.

ERR XTDE0430

It is a [dynamic error](#) if the 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 [dynamic error](#) if the 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 XTDE0450](#)

It is a [dynamic error](#) if the result sequence contains a function item.

[ERR XTDE0540](#)

It is a [dynamic error](#) if the conflict resolution algorithm for template rules leaves more than one matching template rule when the declaration of the relevant [mode](#) has an [on-multiple-match](#) attribute with the value [fail](#).

[ERR XTDE0555](#)

It is a [dynamic error](#) if [xsl:apply-templates](#), [xsl:apply-imports](#) or [xsl:next-match](#) is used to process a node using a mode whose declaration specifies [on-no-match="fail"](#) when there is no [template rule](#) in the [stylesheet](#) whose match pattern matches that node.

[ERR XTDE0560](#)

It is a [dynamic error](#) if [xsl:apply-imports](#) or [xsl:next-match](#) is evaluated when the [current template rule](#) is [absent](#).

[ERR XTDE0640](#)

In general, a [circularity](#) in a [stylesheet](#) is a [dynamic error](#).

[ERR XTDE0700](#)

It is a [dynamic error](#) if a template that has an [explicitly mandatory](#) or [implicitly mandatory](#) parameter is invoked without supplying a value for that parameter.

[ERR XTDE0820](#)

It is a [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 [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:element](#) instruction.

[ERR XTDE0835](#)

It is a [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](#) datatype or if it is the string <http://www.w3.org/2000/xmlns/>.

[ERR XTDE0850](#)

It is a [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 [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 [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 [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 datatype or if it is the string <http://www.w3.org/2000/xmlns/>.

[ERR XTDE0890](#)

It is a [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 [dynamic error](#) if the string value of the new namespace node is not valid in the lexical space of the datatype xs:anyURI, or if it is the string <http://www.w3.org/2000/xmlns/>.

[ERR XTDE0920](#)

It is a [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 [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 [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 [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 [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. If the processor is able to detect the error statically, it MAY optionally signal it as a [static error](#).

[ERR XTDE1035](#)

It is a [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 XTDE1061](#)

It is a [dynamic error](#) if the [current-group](#) function is used when the current group is [absent](#), or when it is invoked in the course of evaluating a pattern. The error MAY be reported statically if it can be detected statically.

[ERR XTDE1071](#)

It is a [dynamic error](#) if the [current-grouping-key](#) function is used when the current grouping key is [absent](#), or when it is invoked in the course of evaluating a pattern. The error MAY be reported statically if it can be detected statically.

[ERR XTDE1110](#)

It is a [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 [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 [\[Functions and Operators 3.0\]](#). 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 [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 [\[Functions and Operators 3.0\]](#). 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 XTDE1160](#)

When a URI reference [supplied to the [`document`](#) function] contains a fragment identifier, it is a [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).

[ERR XTDE1162](#)

When a URI reference [supplied to the [`document`](#) function] is a relative reference, it is a [dynamic error](#) if no base URI is available to resolve the relative reference. This can arise for example when the URI is contained in a node that has no base URI (for example a parentless text node), or when the second argument to the function is a node that has no base URI, or when the base URI from the static context is undefined.

[ERR XTDE1260](#)

It is a [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 containing [package](#). 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 [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 XTDE1360](#)

If the [`current`](#) function is evaluated within an expression that is evaluated when the context item is absent, a [dynamic error](#) occurs.

[ERR XTDE1370](#)

It is a [dynamic error](#) if \$node, or the context item if the second argument is omitted, is a node in a tree whose root is not a document node.

[ERR XTDE1380](#)

It is a [dynamic error](#) if \$node, or the context item if the second argument is omitted, is a node in a tree whose root is not a document node.

[ERR XTDE1390](#)

It is a [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 XTDE1400

It is a [dynamic error](#) if the argument [passed to the [function-available](#) function] does not evaluate to a string that is a valid [EQName](#), or if the value is a [lexical QName](#) with a prefix for which no namespace declaration is present in the static context. 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 [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 the containing element is processed with [XSLT 1.0 behavior](#), it is a [dynamic error](#) to evaluate an extension function call if no implementation of the extension function is available.

ERR XTDE1428

It is a [dynamic error](#) if the argument [passed to the [type-available](#) function] does not evaluate to a string that is a valid [EQName](#), or if the value is a [lexical QName](#) with a prefix for which no namespace declaration is present in the static context. 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 [dynamic error](#) if the argument [passed to the [element-available](#) function] does not evaluate to a string that is a valid [EQName](#), or if the value is a [lexical QName](#) with a prefix for which no namespace declaration is present in the static context. 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 [dynamic error](#) if it has no [xsl:fallback](#) children.

ERR XTDE1460

It is a [dynamic error](#) if the [effective value](#) of the [format](#) attribute [of an [xsl:result-document](#) element] is not a valid [EQName](#), or if it does not match the [expanded QName](#) of an [output definition](#) in the containing [package](#). 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 [dynamic error](#) to evaluate the [xsl:result-document](#) instruction in [temporary output state](#).

ERR XTDE1490

It is a [dynamic error](#) for a transformation to generate two or more [final result trees](#) with the same URI.

ERR XTDE1500

It is a [dynamic error](#) for a [stylesheet](#) to write to an external resource and read from the same resource during a single transformation, if the same absolute URI is used to access the resource in both cases.

ERR XTDE1665

A [dynamic error](#) MAY be raised if the input to the processor includes an item that requires availability of an optional feature that the processor does not provide.

[ERR XTDE2210](#)

It is a [dynamic error](#) if there are two [`xsl:merge-key`](#) elements that occupy corresponding positions among the [`xsl:merge-key`](#) children of two different [`xsl:merge-source`](#) elements and that have differing [effective values](#) for any of the attributes `lang`, `order`, `collation`, `case-order`, or `data-type`. Values are considered to differ if the attribute is present on one element and not on the other, or if it is present on both elements with [effective values](#) that are not equal to each other. In the case of the `collation` attribute, the values are compared as absolute URIs after resolving against the base URI. The error MAY be reported statically if it is detected statically.

[ERR XTDE2220](#)

It is a [dynamic error](#) if any input sequence to an [`xsl:merge`](#) instruction contains two items that are not correctly sorted according to the merge key values defined on the [`xsl:merge-key`](#) children of the corresponding [`xsl:merge-source`](#) element, when compared using the collation rules defined by the attributes of the corresponding [`xsl:merge-key`](#) children of the [`xsl:merge`](#) instruction, unless the attribute `sort-before-merge` is present with the value `yes`.

[ERR XTDE3052](#)

It is a [dynamic error](#) if an invocation of an abstract component is evaluated.

[ERR XTSE3155](#)

It is a static error if an [`xsl:function`](#) element with no [`xsl:param`](#) children has a `streamability` attribute with any value other than `unclassified`.

[ERR XTDE3160](#)

It is a [dynamic error](#) if the [target expression](#) [of an [`xsl:evaluate`](#) instruction] is not a valid [expression](#) (that is, if a static error occurs when analyzing the string according to the rules of the XPath specification).

[ERR XTDE3175](#)

It is a [dynamic error](#) if an [`xsl:evaluate`](#) instruction is evaluated when use of [`xsl:evaluate`](#) has been statically or dynamically disabled.

[ERR XTDE3240](#)

It is a [dynamic error](#) if the value of `$input` does not conform to the JSON grammar as defined by [\[RFC 7159\]](#), allowing implementation-defined extensions if the `liberal` option is set to `yes`.

[ERR XTDE3245](#)

It is a [dynamic error](#) if the value of the `validate` option is `true` and the processor is not schema-aware.

[ERR XTDE3250](#)

It is a [dynamic error](#) if the value of `$input` contains an escaped representation of a character (or codepoint) that is not a valid character in the version of XML supported by the implementation, unless the `unescape` option is set to `false`.

[ERR XTDE3260](#)

It is a [dynamic error](#) if the value of `$options` includes an entry whose key is `liberal`, `validate`, `unescape`, or `fallback`, and whose value is not a permitted value for that key.

[ERR XTDE3340](#)

It is a [dynamic error](#) if the value of the first argument to the [`accumulator-before`](#) or [`accumulator-after`](#) function is not a valid [EQName](#), 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:accumulator](#) declaration appearing in the [package](#) in which the function call appears. 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 XTDE3350](#)

It is a [dynamic error](#) to call the [accumulator-before](#) or [accumulator-after](#) function when there is no [context item](#).

[ERR XTTE3360](#)

It is a [type error](#) to call the [accumulator-before](#) or [accumulator-after](#) function when the [context item](#) is not a node, or when it is an attribute or namespace node.

[ERR XTDE3362](#)

It is a [dynamic error](#) to call the [accumulator-before](#) or [accumulator-after](#) function when the context item is a node in a tree to which the selected accumulator is not applicable (including the case where it is not applicable because the document is streamed and the accumulator is not declared with `streamable="yes"`). Implementations MAY raise this error but are NOT REQUIRED to do so, if they are capable of streaming documents without imposing this restriction.

[ERR XTDE3365](#)

A [dynamic error](#) occurs if the set of keys in the maps resulting from evaluating the sequence constructor [within an [xsl:map](#) instruction] contains duplicates.

[ERR XTDE3400](#)

It is an error if there is a cyclic set of dependencies among accumulators such that the (pre- or post-descent) value of an accumulator depends directly or indirectly on itself. A processor MAY report this as a [static error](#) if it can be detected statically. Alternatively a processor MAY report this as a [dynamic error](#). As a further option, a processor may fail catastrophically when this error occurs.

[ERR XTDE3480](#)

It is a [dynamic error](#) if the [current-merge-group](#) function is used when the current merge group is [absent](#). The error MAY be reported statically if it can be detected statically.

[ERR XTDE3490](#)

It is a [dynamic error](#) if the `$source` argument of the [current-merge-group](#) function does not match the `name` attribute of any [xsl:merge-source](#) element for the current merge operation. The error MAY be reported statically if it can be detected statically.

[ERR XTDE3510](#)

It is a [dynamic error](#) if the [current-merge-key](#) function is used when the current merge key is [absent](#), or when it is invoked in the course of evaluating a pattern. The error MAY be reported statically if it can be detected statically.

[ERR XTDE3530](#)

It is a [dynamic error](#) if an [xsl:try](#) instruction is unable to recover the state of a final result tree because recovery has been disabled by use of the attribute `rollback-output="no"`.

[ERR XTMM9000](#)

When a transformation is terminated by use of `<xsl:message terminate="yes"/>`, the effect is the same as when a [dynamic error](#) occurs during the transformation. The default error code is XTMM9000; this may be overridden using the `error-code` attribute of the [xsl:message](#) instruction.

[ERR XTMM9001](#)

When a transformation is terminated by use of [xsl:assert](#), the effect is the same as when a [dynamic error](#) occurs during the transformation. The default error code is XTMM9001; this may be overridden using the `error-code` attribute of the [xsl:assert](#) instruction.

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 [27 Conformance](#)) require vendors to provide documentation that explains how these choices have been exercised.

The implementation-defined features are grouped into categories for convenience.

F.1 Application Programming Interfaces

This category covers interfaces for initiating a transformation, setting its parameters, initializing the static and dynamic context, and collecting the results. In general terms, it is implementation defined how input is passed to the processor and how it returns its output. This includes the interpretation of URIs used to refer to stylesheet packages and modules, source documents and collections, collations, and result documents.

More specifically:

1. If the initialization of any [global variables](#) or [parameter](#) depends on the context item, a dynamic error can occur if the context item is absent. It is implementation-defined whether this error occurs during priming of the stylesheet or subsequently when the variable is referenced; and it is implementation-defined whether the error occurs at all if the variable or parameter is never referenced. (See [2.3.2 Priming a Stylesheet](#))
2. 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 [2.3.2 Priming a Stylesheet](#))
3. The way in which a [base output URI](#) is established is implementation-defined (See [2.3.6.2 Serializing the Result](#))
4. It is implementation-defined how a package is located given its name and version, and which version of a package is chosen if several are available. (See [3.5.2 Dependencies between Packages](#))
5. In the absence of an `[xsl:]default-collation` attribute, the default collation `MAY` be set by the calling application in an implementation-defined way. (See [3.7.1 The default-collation Attribute](#))
6. 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. 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 also implementation-defined. (See [3.11.1 Locating Stylesheet Modules](#))
7. The [statically known documents](#)^{XP30}, [statically known collections](#)^{XP30}, and the [statically known default collection type](#)^{XP30} are implementation-defined. (See [5.3.1 Initializing the Static Context](#))
8. Implementations may provide user options that relax the requirement for the [doc](#)^{FO30} and [collection](#)^{FO30} functions (and therefore, by implication, the [document](#) function) to return stable results. The manner in which

- such user options are provided, if at all, is implementation-defined. (See [5.3.3 Initializing the Dynamic Context](#))
9. Streamed processing may be initiated by invoking the transformation with an [initial mode](#) declared as streamable, while supplying the [initial match selection](#) (in an implementation-defined way) as a streamed document. (See [6.6.4 Streamable Templates](#))
 10. The mechanism by which the caller supplies a value for a [stylesheet parameter](#) is implementation-defined. (See [9.5 Global Variables and Parameters](#))
 11. The detail of any external mechanism allowing a processor to enable or disable checking of assertions is implementation-defined. (See [23.2 Assertions](#))
 12. The way in which the results of the transformation are delivered to an application is implementation-defined. (See [25 Transformation Results](#))
 13. It is implementation-defined how the URI appearing in the `href` attribute of [xsl:result-document](#) affects the way in which the result tree is delivered to the application. There `MAY` be restrictions on the form of this URI. (See [25.1 Creating Secondary Results](#))
 14. If serialization is supported, then the location to which a [final result tree](#) is serialized is implementation-defined, subject to the constraint that relative URI references used to reference one tree from another remain valid. (See [26 Serialization](#))

[F.2 Vendor and User Extensions](#)

This category covers extensions and extensibility: mechanisms for providing vendor or user extensions to the language without sacrificing interoperability.

In general terms, it is implementation-defined:

- whether and under what circumstances the implementation recognizes any extension functions, extension instructions, extension attributes, user-defined data elements, additional types, additional serialization methods or serialization parameters, or additional collations, and if so, what effect they have.
- whether, how, and under what circumstances the implementation allows users to define extension functions, extension instructions, extension attributes, user-defined data elements, additional types, additional serialization methods or serialization parameters, or additional collations. If it does allow users to do so, it must follow the rules given elsewhere in this specification.
- what information is available to such extensions (for example, whether they have access to the static and dynamic context.)
- where such extensions are allowed, the extent to which the processor enforces their correct behavior (for example, checking that strings returned by extension functions contain only valid XML characters)

More specifically:

1. The mechanisms for creating new [extension instructions](#) and [extension functions](#) are implementation-defined. It is not `REQUIRED` that implementations provide any such mechanism. (See [2.9 Extensibility](#))
2. 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 [3.7.3 User-defined Data Elements](#))
3. The effect of user-defined data elements whose name is in a namespace recognized by the implementation is implementation-defined. (See [3.7.3 User-defined Data Elements](#))

4. 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 [3.14 Built-in Types](#))
5. The set of extension functions available in the static context for the target expression of [`xsl:evaluate`](#) is implementation-defined. (See [10.4.1 Static context for the target expression](#))
6. 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](#))
7. The [posture](#) and [sweep](#) of [extension functions \(and references to extension functions\)](#) and [extension instructions](#) are implementation-defined. (See [19.8.4.2 Streamability of extension instructions](#))
8. Additional [streamability categories](#) for stylesheet functions may be defined by an implementation. (See [19.8.5 Classifying Stylesheet Functions](#))
9. The effect of an extension function returning a string containing characters that are not permitted in XML is implementation-defined. (See [24.1.2 Calling Extension Functions](#))
10. The way in which external objects are represented in the type system is implementation-defined. (See [24.1.3 External Objects](#))

F.3 Localization

This specification, and the specifications that it refers to, include facilities for adapting the output of a transformation to meet local expectations: examples include the formatting of numbers and dates, and the choice of collations for sorted output. The general principles are:

- The specification does not mandate any particular localizations that processors must offer: for example, a conformant processor might choose to provide output in Japanese only.
- The specification provides fallback mechanisms so that if a particular localization is requested and is not available, processing does not fail.

More specifically:

1. The combinations of languages and numbering sequences recognized by the [`xsl:number`](#) instruction, beyond those defined as mandatory in this specification, are implementation-defined. There `MAY` be implementation-defined upper bounds on the numbers that can be formatted using any particular numbering sequence. There `MAY` be constraints on the values of the `ordinal` attribute recognized for any given language. (See [12.4 Number to String Conversion Attributes](#))
2. The facilities for defining collations and allocating URIs to identify them are largely implementation-defined. (See [13.1.3 Sorting Using Collations](#))
3. 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](#))
4. If none of the `collation`, `lang`, or `case-order` attributes is present (on [`xsl:sort`](#)), the collation is chosen in an implementation-defined way. (See [13.1.3 Sorting Using Collations](#))
5. When using the family of URIs that invoke the Unicode Collation Algorithm, the effect of supplying a query keyword or value not defined in this specification is implementation-defined. The defaults for query keywords are also implementation-defined unless otherwise stated. (See [13.4 The Unicode Collation Algorithm](#))

F.4 Optional Features

As well as the optional conformance features identified in [27 Conformance](#), some specific features of the specification are defined to be optional.

1. It is implementation-defined whether an XSLT 3.0 processor supports backwards compatible behavior for any XSLT version earlier than XSLT 3.0. (See [3.9 Backwards Compatible Processing](#))
2. 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](#))
3. It is implementation-defined whether, and under what circumstances, disabling output escaping is supported. (See [26.2 Disabling Output Escaping](#))

F.5 Dependencies

When this specification refers normatively to other specifications, it generally gives implementations freedom to decide (within constraints) which version of the referenced specification should be used. Specifically:

1. It is implementation-defined which versions and editions of XML and XML Namespaces (1.0 and/or 1.1) are supported. (See [4.1 XML Versions](#))
2. It is implementation-defined which versions of XML, HTML, and XHTML are supported in the `version` attribute of the `xsl:output` declaration. (See [26 Serialization](#))
3. It is implementation-defined whether (and if so how) an XSLT 3.0 processor is able to work with versions of XPath later than XPath 3.1. (See [27 Conformance](#))
4. It is implementation-defined whether (and if so how) an XSLT 3.0 processor is able to work with versions of [\[XSLT and XQuery Serialization\]](#) later than 3.1. (See [27.3 Serialization Feature](#))

F.6 Defaults and Limits

To accommodate variations in the way that the XSLT language is deployed, and the constraints of different processing environments, defaults for some options are implementation-defined. In addition, limits on the sizes of ranges of values permitted are in general implementation-defined:

1. Limits on the value space of primitive datatypes, where not fixed by [\[XML Schema Part 2\]](#), are implementation-defined. (See [4.7 Limits](#))
2. The default value of the `encoding` attribute of the `xsl:output` element is implementation-defined. Where the encoding is UTF-8, the default for the `byte-order-mark` attribute is implementation-defined. (See [26 Serialization](#))

F.7 Detection and Reporting of Errors

Some aspects of error handling are implementation-defined:

1. It is implementation-defined whether type errors are signaled statically. (See [2.14 Error Handling](#))

2. If the [effective version](#) of any element in the stylesheet is not 1.0 or 2.0 but is less than 3.0, the RECOMMENDED action is to report a static error; however, processors MAY recognize such values and process the element in an implementation-defined way. (See [3.9 Backwards Compatible Processing](#))
3. The default values for the `warning-on-no-match` and `warning-on-multiple-match` attributes of `xsl:mode` are implementation-defined. (See [6.6.1 Declaring Modes](#))
4. The form of any warnings output when there is no matching template rule, or when there are multiple matching template rules, is implementation-defined. (See [6.6.1 Declaring Modes](#))
5. The destination and formatting of messages written using the `xsl:message` instruction are implementation-defined. (See [23.1 Messages](#))

G Summary of Available Functions (Non-Normative)

G.1 Function Classification

The functions available for use within an XSLT stylesheet can be classified based firstly, on where the function is defined, and secondly, on where it can be used. Specifically, the set of functions available is slightly different for :

- Regular XPath expressions within the stylesheet, for example those appearing in `select` or `test` attributes, or between braces in a [text value template](#) (*R*)
- [Static expressions](#) (*S*)
- XPath expressions evaluated dynamically using [xsl:evaluate](#) (*D*)

The categories are listed in the following table:

Categories of Function, and their Availability

Category	Defined where?	Available where?	Notes
User-defined functions	Defined using xsl:function declarations in the stylesheet	<i>R, D</i>	Functions are private by default; private functions can be referenced only within the package where they are declared (and not in xsl:evaluate expressions).
Constructor functions for built-in types	Section 17 Constructor functions ^{FO30}	<i>R, S, D</i>	These functions are all in the namespace conventionally associated with the prefix <code>xs</code> . The semantics of a constructor function are identical to the semantics of a <code>cast</code> expression.
Constructor functions for user-defined types	Section 17 Constructor functions ^{FO30}	<i>R, D (if schema-aware="yes")</i>	This category includes a function for every named user-defined simple type in an imported schema; the function allows the conversion of strings and certain other values to instances of the user-defined type.

Category	Defined where?	Available where?	Notes
Functions defined in XPath 3.0	[Functions and Operators 3.0]	R, S, D	Includes functions in the namespaces conventionally referred to be the prefixes <code>fn</code> and <code>math</code> .
Additional functions defined in XPath 3.1 (where supported)	[Functions and Operators 3.1]	R, S, D.	This category has an overlap with the set of XSLT-defined-functions. Where a function is defined both in this document and in XPath 3.1, the function is available in an XSLT 3.0 stylesheet whether or not the processor supports XPath 3.1. This category includes functions in namespaces conventionally referred to by the prefixes <code>fn</code> , <code>map</code> , and <code>array</code> .
Functions defined in XSLT 3.0	This specification	R, S (see note), D	See G.2 List of XSLT-defined functions . There is an overlap with the set of functions defined in XPath 3.1. The functions available in static expressions are: <code>element-available</code> , <code>function-available</code> , <code>type-available</code> , <code>available-system-properties</code> , and <code>system-property</code> .
Extension functions	Implementation-defined: see 24.1 Extension Functions .	R, S, D	Availability is <code>implementation-defined</code>

G.2 [List of XSLT-defined functions](#)

This appendix acts as an index of functions defined in this specification, to augment the set of functions defined in [\[Functions and Operators 3.0\]](#).

accumulator-after

See [18.2.7 fn:accumulator-after](#)

accumulator-before

See [18.2.6 fn:accumulator-before](#)

available-system-properties

See [20.4.5 fn:available-system-properties](#)

collation-key

See [21.2.12 fn:collation-key](#)

copy-of

See [18.3 fn:copy-of](#)

current

See [20.4.1 fn:current](#)

current-group

See [14.2.1 fn:current-group](#)

current-grouping-key

See [14.2.2 fn:current-grouping-key](#)

current-merge-group

See [15.6.1 fn:current-merge-group](#)

current-merge-key

See [15.6.2 fn:current-merge-key](#)

current-output-uri

See [25.3.1 fn:current-output-uri](#)

document

See [20.1 fn:document](#)

element-available

See [24.2.2 fn:element-available](#)

function-available

See [24.1.1 fn:function-available](#)

json-to-xml

See [22.3 fn:json-to-xml](#)

key

See [20.2.2 fn:key](#)

map:contains

See [21.2.5 map:contains](#)

map:entry

See [21.2.8 map:entry](#)

map:find

See [21.2.11 map:find](#)

map:for-each

See [21.2.10 map:for-each](#)

map:get

See [21.2.6 map:get](#)

map:keys

See [21.2.4 map:keys](#)

map:merge

See [21.2.2 map:merge](#)

map:put

See [21.2.7 map:put](#)

map:remove

See [21.2.9 map:remove](#)

map:size

See [21.2.3 map:size](#)

regex-group

See [17.2 fn:regex-group](#)

snapshot

See [18.4 fn:snapshot](#)

stream-available

See [18.1.3 fn:stream-available](#)

system-property

See [20.4.4 fn:system-property](#)

type-available

See [24.1.4 fn:type-available](#)

unparsed-entity-public-id

See [20.4.3 fn:unparsed-entity-public-id](#)

unparsed-entity-uri

See [20.4.2 fn:unparsed-entity-uri](#)

xml-to-json

See [22.4 fn:xml-to-json](#)

H Schemas for XSLT 3.0 Stylesheets (Non-Normative)

For convenience, schemas are provided for validation of XSLT 3.0 stylesheets using the XSD 1.1 and Relax NG schema languages. These are non-normative. Neither will detect every static error that might arise in an XSLT 3.0 stylesheet (for example, there is no attempt to check the syntax of XPath expressions); in addition, these schemas may reject some stylesheets that are valid, for example because they rely on `xsl:use-when` to eliminate sections of code that would otherwise be invalid.

H.1 XSD 1.1 Schema for XSLT Stylesheets

The following XSD 1.1 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 datatype 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 [schema-for-xslt30.xsd](#)

Note:

The schema as written uses a lax wildcard to permit literal result elements to appear in a sequence constructor. This assumes that the schema used for validation will not contain any global element declaration that matches the element name of a literal result element. The content model for an element such as `invoice` appearing within a stylesheet is not the same as the content model for the same element appearing within a source document (it is likely to contain XSLT instructions rather than other elements from the target vocabulary): therefore, including such declarations in the schema used for validating a stylesheet is inappropriate.

The reason that lax validation rather than skip validation is used is so that XSLT instructions appearing as children of the literal result element will themselves be validated, using the appropriate global element declaration.

Note:

The schema uses XSD 1.1 assertions to represent some of the non-grammatical constraints appearing in the specification, for example the rule that some elements can have either a `select` attribute or a contained sequence constructor, but not both. At this stage, no attempt has been made to represent every such constraint, even where it is not difficult to express the rule. There will always be some constraints that cannot be expressed at all, for example those that require access to multiple stylesheet modules, those that require access to the in-scope schema components, and those that involve parsing a non-regular grammar, such as the grammar for patterns.

Apart from assertions, the only other significant use of XSD 1.1 features is that the elements `xsl:param` and `xsl:variable` are in two substitution groups: one containing all instructions, and one containing all declarations. If the schema needs to be converted to an XSD 1.0 schema, removing all assertions is straightforward; the other change needed is to remove `xsl:param` and `xsl:variable` from the substitution group for declarations, and instead permit them explicitly as children of `xsl:transform`.

```

<?xml version="1.0" encoding="UTF-8"?>
<!--* <!DOCTYPE xs:schema PUBLIC "-//W3C//DTD XMLSCHEMA 200105//EN"
           "http://www.w3.org/2001/XMLSchema.dtd" [
<!ENTITY % schemaAttrs "
  xmlns:xs   CDATA #IMPLIED
  xmlns:xsl  CDATA #IMPLIED
  xmlns:xsd  CDATA #IMPLIED"
>
<!ENTITY % p "xs:">
<!ENTITY % s ":xs">
]>*-->
<?xmlstylesheet href="http://www.w3.org/2008/09/xsd.xsl" type="text/xsl"?>
<xss: schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
               xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
               xmlns:vc="http://www.w3.org/2007/XMLSchema-versioning"
               targetNamespace="http://www.w3.org/1999/XSL/Transform"
               elementFormDefault="qualified"
               vc:minVersion="1.1">

<!-- ++++++ -->
<xss:annotation>
  <xss:documentation>
    <p>
      This is an XSD 1.1 schema for XSLT 3.0 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.
    </p>
    <p>
      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
    </p>
    <p>
      The schema is organized as follows:
    </p>
    <ul>
      <li>
        PART A: definitions of complex types and model groups used as the basis
        for element definitions
      </li>
      <li>
        PART B: definitions of individual XSLT elements
      </li>
      <li>
        PART C: definitions for literal result elements
      </li>
      <li>
        PART D: definitions of simple types used in attribute definitions
      </li>
    </ul>
    <p>
      The schema has a number of limitations:
    </p>
    <ul>
      <li>
        The XSLT specification allows additional elements and attributes to be
        present where forwards compatibility is invoked. This schema does not.
      </li>
      <li>
        The XSLT specification allows arbitrary content in a part of the

```

```

stylesheet that is excluded by virtue of a use-when attribute. This
schema does not.

</li>
<li>
  The handling of literal result elements in this schema is imperfect;
  although various options are allowed, none matches the specification
  exactly. For example, the content of a literal result element uses lax
  validation, which permits child elements in the XSLT namespace that have
  no declaration in this schema.

</li>
<li>
  The schema makes no attempt to check XPath expressions for syntactic or
  semantic correctness, nor to check that component references are
  resolved (for example that a template named in xsl:call-template has a
  declaration). Doing this in general requires cross-document validation,
  which is beyond the scope of XSD.

</li>
<li>
  The schema imports the schema for XSD 1.0 schema documents. In
  stylesheets that contain an inline XSD 1.1 schema, this import should be
  replaced with one for the schema for XSD 1.1 schema documents.

</li>
</ul>
</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. We use the XSD 1.1 version.
-->

<xs:import namespace="http://www.w3.org/2001/XMLSchema"
            schemaLocation="http://www.w3.org/TR/xmlschema11-1/XMLSchema.xsd"/>

<!-- ++++++ -->
<xs:annotation>
  <xs:documentation>
    <p>
      PART A: definitions of complex types and model groups used as the basis
      for element definitions
    </p>
  </xs:documentation>
</xs:annotation>
<!-- ++++++ -->

<xs:complexType name="generic-element-type" mixed="true">
  <xs:annotation>
    <xs:documentation>
      <p>
        This complex type provides a generic supertype for all XSLT elements; it
        contains the definitions of the standard attributes that may appear on
        any element.
      </p>
    </xs:documentation>
  </xs:annotation>

```

```

        </p>
    </xs:documentation>
</xs:annotation>
<xs:attribute name="default-collation" type="xsl:uri-list"/>
<xs:attribute name="default-mode" type="xsl:default-mode-type"/>
<xs:attribute name="default-validation"
              type="xsl:validation-strip-or-preserve"
              default="strip"/>
<xs:attribute name="exclude-result-prefixes" type="xsl:prefix-list-or-all"/>
<xs:attribute name="expand-text" type="xsl:yes-or-no"/>
<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:attribute name="_default-collation" type="xs:string"/>
<xs:attribute name="_default-mode" type="xs:string"/>
<xs:attribute name="_default-validation" type="xs:string"/>
<xs:attribute name="_exclude-result-prefixes" type="xs:string"/>
<xs:attribute name="_expand-text" type="xs:string"/>
<xs:attribute name="_extension-element-prefixes" type="xs:string"/>
<xs:attribute name="_use-when" type="xs:string"/>
<xs:attribute name="_xpath-default-namespace" type="xs:string"/>
<xs:anyAttribute namespace="#other" processContents="lax"/>
</xs:complexType>

<xs:complexType name="versioned-element-type" mixed="true">
    <xs:annotation>
        <xs:documentation>
            <p>
                This complex type provides a generic supertype for all XSLT elements
                with the exception of xsl:output; it contains the definitions of the
                version attribute that may appear on any element.
            </p>
            <p>
                The xsl:output does not use this definition because, although it has a
                version attribute, the syntax and semantics of this attribute are
                unrelated to the standard version attribute allowed on other elements.
            </p>
        </xs:documentation>
    </xs:annotation>
    <xs:complexContent>
        <xs:extension base="xsl:generic-element-type">
            <xs:attribute name="version" type="xs:decimal" use="optional"/>
            <xs:attribute name="_version" type="xs:string"/>
        </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:annotation>
        <xs:documentation>
            <p>
                This complex type provides a generic supertype for all XSLT elements
                that allow a sequence constructor as their content.
            </p>
        </xs:documentation>
    </xs:annotation>

```

```

        </p>
    </xs:documentation>
</xs:annotation>
<xs:complexContent mixed="true">
    <xs:extension base="xsl:versioned-element-type">
        <xs:group ref="xsl:sequence-constructor-group"
            min0Occurs="0"
            max0Occurs="unbounded"/>
    </xs:extension>
</xs:complexContent>
</xs:complexType>

<xs:complexType name="sequence-constructor-and-select">
    <xs:annotation>
        <xs:documentation>
            <p>
                This complex type allows a sequence constructor and a select attribute.
            </p>
        </xs:documentation>
    </xs:annotation>
    <xs:complexContent mixed="true">
        <xs:extension base="xsl:sequence-constructor">
            <xs:attribute name="select" type="xsl:expression"/>
            <xs:attribute name="_select" type="xs:string"/>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>

<xs:complexType name="sequence-constructor-or-select">
    <xs:annotation>
        <xs:documentation>
            <p>
                This complex type allows a sequence constructor or a select attribute,
                but not both.
            </p>
        </xs:documentation>
    </xs:annotation>
    <xs:complexContent mixed="true">
        <xs:restriction base="xsl:sequence-constructor-and-select">
            <xs:group ref="xsl:sequence-constructor-group"
                min0Occurs="0"
                max0Occurs="unbounded"/>
            <xs:anyAttribute namespace="#other" processContents="lax"/>
            <xs:assert test="not(exists(@select | @_select) and
                (exists(* except xsl:fallback) or exists(text()[normalize-space()])))" />
        </xs:restriction>
    </xs:complexContent>
</xs:complexType>

<xs:group name="sequence-constructor-group">
    <xs:annotation>
        <xs:documentation>
            <p>
                This complex type provides a generic supertype for all XSLT elements
                that allow a sequence constructor as their content.
            </p>
        </xs:documentation>
    </xs:annotation>
    <xs:choice>
        <xs:element ref="xsl:instruction"/>
        <xs:group ref="xsl:result-elements"/>
    </xs:choice>
</xs:group>

```

```

</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>
    <p>
      PART B: definitions of individual XSLT elements Elements are listed in
      alphabetical order.
    </p>
  </xs:documentation>
</xs:annotation>
<!-- ++++++ -->

<xs:element name="accept">
  <xs:annotation>
    <xs:documentation>
      <p>
        This element appears as a child of xsl:use-package and defines any
        variations that the containing package wishes to make to the visibility
        of components made available from a library package. For example, it may
        indicate that some of the public components in the library package are
        not to be made available to the containing package.
      </p>
    </xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="xsl:element-only-versioned-element-type">
        <xs:attribute name="component" type="xsl:component-kind-type"/>
        <xs:attribute name="names" type="xsl:EQNames"/>
        <xs:attribute name="visibility" type="xsl:visibility-type"/>
        <xs:attribute name="_component" type="xs:string"/>
        <xs:attribute name="_names" type="xs:string"/>
        <xs:attribute name="_visibility" type="xs:string"/>
        <xs:assert test="exists(@component | @_component)"/>
        <xs:assert test="exists(@names | @_names)"/>
        <xs:assert test="exists(@visibility | @_visibility)"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="accumulator" substitutionGroup="xsl:declaration">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="xsl:element-only-versioned-element-type">
        <xs:sequence>
          <xs:element ref="xsl:accumulator-rule" minOccurs="1" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="name" type="xsl:QName"/>
        <xs:attribute name="initial-value" type="xsl:expression"/>
        <xs:attribute name="as" type="xsl:sequence-type"/>
        <xs:attribute name="streamable" type="xsl:yes-or-no"/>
        <xs:attribute name="_name" type="xs:string"/>
        <xs:attribute name="_initial-value" type="xs:string"/>
        <xs:attribute name="_as" type="xs:string"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

```

```

        <xs:attribute name="_streamable" type="xs:string"/>
    </xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>

<xs:element name="accumulator-rule">
<xs:complexType>
<xs:complexContent mixed="true">
<xs:extension base="xsl:sequence-constructor-or-select">
<xs:sequence/>
<xs:attribute name="match" type="xsl:pattern"/>
<xs:attribute name="phase">
<xs:simpleType>
<xs:restriction base="xs:token">
<xs:enumeration value="start"/>
<xs:enumeration value="end"/>
</xs:restriction>
</xs:simpleType>
</xs:attribute>
<xs:attribute name="_match" type="xs:string"/>
<xs:attribute name="_phase" type="xs:string"/>
<xs:assert test="exists(@match | @_match)"/>
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>

<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"/>
<xs:attribute name="regex" type="xsl:avt"/>
<xs:attribute name="flags" type="xsl:avt" default="" />
<xs:attribute name="_select" type="xs:string"/>
<xs:attribute name="_regex" type="xs:string"/>
<xs:attribute name="_flags" type="xs:string"/>
<xs:assert test="exists(@select | @_select)"/>
<xs:assert test="exists(@regex | @_regex)"/>
</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:attribute name="_select" type="xs:string"/>
        <xs:attribute name="_mode" type="xs:string"/>
        <xs:assert test="every $e in subsequence(xsl:sort, 2)
                        satisfies empty($e/(@stable | @_stable))">
          <xs:annotation>
            <xs:documentation>
              <p>
                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.
              </p>
            </xs:documentation>
          </xs:annotation>
        </xs:assert>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="assert" substitutionGroup="xsl:instruction">
  <xs:complexType>
    <xs:complexContent mixed="true">
      <xs:extension base="xsl:sequence-constructor">
        <xs:attribute name="test" type="xsl:expression"/>
        <xs:attribute name="select" type="xsl:expression"/>
        <xs:attribute name="error-code" type="xsl:avt"/>
        <xs:attribute name="_test" type="xs:string"/>
        <xs:attribute name="_select" type="xs:string"/>
        <xs:attribute name="_error-code" type="xs:string"/>
        <xs:assert test="exists(@test | @_test)"/>
      </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-or-select">
        <xs:attribute name="name" type="xsl:avt"/>
        <xs:attribute name="namespace" type="xsl:avt"/>
        <xs:attribute name="separator" type="xsl:avt"/>
        <xs:attribute name="type" type="xsl:QName"/>
        <xs:attribute name="validation" type="xsl:validation-type"/>
        <xs:attribute name="_name" type="xs:string"/>
        <xs:attribute name="_namespace" type="xs:string"/>
        <xs:attribute name="_separator" type="xs:string"/>
        <xs:attribute name="_type" type="xs:string"/>
        <xs:attribute name="_validation" type="xs:string"/>
        <xs:assert test="not(exists(@type | @_type) and exists(@validation | @_validation))">

```

```

<xs:annotation>
  <xs:documentation>
    <p>
      The type and validation attributes are mutually exclusive (if
      one is present, the other must be absent).
    </p>
  </xs:documentation>
</xs:annotation>
</xs:assert>
<xs:assert test="exists(@name | @_name)"/>
</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"/>
        <xs:attribute name="streamable" type="xsl:yes-or-no"/>
        <xs:attribute name="use-attribute-sets" type="xsl:QNames" default="" />
        <xs:attribute name="visibility" type="xsl:visibility-type"/>
        <xs:attribute name="_name" type="xs:string"/>
        <xs:attribute name="_streamable" type="xs:string"/>
        <xs:attribute name="_use-attribute-sets" type="xs:string"/>
        <xs:attribute name="_visibility" type="xs:string"/>
        <xs:assert test="exists(@name | @_name)"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="break"
  substitutionGroup="xsl:instruction"
  type="xsl:sequence-constructor-or-select"/>

<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"/>
        <xs:attribute name="_name" type="xs:string"/>
        <xs:assert test="exists(@name | @_name)"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="catch">
  <xs:complexType>
    <xs:complexContent mixed="true">
      <xs:extension base="xsl:sequence-constructor-or-select">
        <xs:attribute name="errors" type="xs:token" use="optional"/>
        <xs:attribute name="_errors" type="xs:string"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

```

```

        </xs:extension>
    </xs:complexContent>
</xs:complexType>
</xs:element>

<xs:element name="character-map" substitutionGroup="xsl:declaration">
    <xs:complexType>
        <xs:complexContent 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"/>
            <xs:attribute name="use-character-maps" type="xsl:QNames" default="" />
            <xs:attribute name="_name" type="xs:string"/>
            <xs:attribute name="_use-character-maps" type="xs:string"/>
            <xs:assert test="exists(@name | @_name)"/>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
</xs:element>

<xs:element name="choose" substitutionGroup="xsl:instruction">
    <xs:complexType>
        <xs:complexContent 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"
            type="xsl:sequence-constructor-or-select"/>

<xs:element name="context-item">
    <xs:complexType>
        <xs:complexContent base="xsl:element-only-versioned-element-type">
            <xs:attribute name="as" type="xsl:item-type"/>
            <xs:attribute name="use">
                <xs:simpleType>
                    <xs:restriction base="xs:token">
                        <xs:enumeration value="required"/>
                        <xs:enumeration value="optional"/>
                        <xs:enumeration value="absent"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:attribute>
            <xs:attribute name="_as" type="xs:string"/>
            <xs:attribute name="_use" type="xs:string"/>
        </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="select" type="xsl:expression"/>
      <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:EQNames" default="" />
      <xs:attribute name="type" type="xsl:QName"/>
      <xs:attribute name="validation" type="xsl:validation-type"/>
      <xs:attribute name="_select" type="xs:string"/>
      <xs:attribute name="_copy-namespaces" type="xs:string"/>
      <xs:attribute name="_inherit-namespaces" type="xs:string"/>
      <xs:attribute name="_use-attribute-sets" type="xs:string"/>
      <xs:attribute name="_type" type="xs:string"/>
      <xs:attribute name="_validation" type="xs:string"/>
      <xs:assert test="not(exists(@type | @_type) and exists(@validation | @_validation))">
        <xs:annotation>
          <xs:documentation>
            <p>
              The type and validation attributes are mutually exclusive (if
              one is present, the other must be absent).
            </p>
          </xs:documentation>
        </xs:annotation>
      </xs:assert>
    </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"/>
        <xs:attribute name="copy-accumulators" type="xsl:yes-or-no" default="no"/>
        <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:attribute name="_select" type="xs:string"/>
        <xs:attribute name="_copy-accumulators" type="xs:string"/>
        <xs:attribute name="_copy-namespaces" type="xs:string"/>
        <xs:attribute name="_type" type="xs:string"/>
        <xs:attribute name="_validation" type="xs:string"/>
        <xs:assert test="not(exists(@type | @_type) and exists(@validation | @_validation))">
          <xs:annotation>
            <xs:documentation>
              <p>
                The type and validation attributes are mutually exclusive (if
                one is present, the other must be absent).
              </p>
            </xs:documentation>
          </xs:annotation>
        </xs:assert>
        <xs:assert test="exists(@select | @_select)" />
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="document" substitutionGroup="xsl:instruction">

```

```

<xs:complexType>
  <xs:complexContent mixed="true">
    <xs:extension base="xsl:sequence-constructor">
      <xs:attribute name="type" type="xsl:QName"/>
      <xs:attribute name="validation" type="xsl:validation-type"/>
      <xs:attribute name="_type" type="xs:string"/>
      <xs:attribute name="_validation" type="xs:string"/>
      <xs:assert test="not(exists(@type | @_type) and exists(@validation | @_validation))">
        <xs:annotation>
          <xs:documentation>
            <p>
              The type and validation attributes are mutually exclusive (if
              one is present, the other must be absent).
            </p>
          </xs:documentation>
        </xs:annotation>
      </xs:assert>
    </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="exponent-separator" type="xsl:char" default="e"/>
        <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="~"/>
        <xs:attribute name="zero-digit" type="xsl:zero-digit" default="0"/>
        <xs:attribute name="digit" type="xsl:char" default="#">
        <xs:attribute name="pattern-separator" type="xsl:char" default="; "/>
        <xs:attribute name="_name" type="xs:string"/>
        <xs:attribute name="_decimal-separator" type="xs:string"/>
        <xs:attribute name="_grouping-separator" type="xs:string"/>
        <xs:attribute name="_infinity" type="xs:string"/>
        <xs:attribute name="_minus-sign" type="xs:string"/>
        <xs:attribute name="_exponent-separator" type="xs:string"/>
        <xs:attribute name="_NaN" type="xs:string"/>
        <xs:attribute name="_percent" type="xs:string"/>
        <xs:attribute name="_per-mille" type="xs:string"/>
        <xs:attribute name="_zero-digit" type="xs:string"/>
        <xs:attribute name="_digit" type="xs:string"/>
        <xs:attribute name="_pattern-separator" type="xs:string"/>
      </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"/>
        <xs:attribute name="namespace" type="xsl:avt"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

```

```

<xs:attribute name="inherit-namespaces" type="xsl:yes-or-no" default="yes"/>
<xs:attribute name="use-attribute-sets" type="xsl:EQNames" default="" />
<xs:attribute name="type" type="xsl:QName"/>
<xs:attribute name="validation" type="xsl:validation-type"/>
<xs:attribute name="_name" type="xs:string"/>
<xs:attribute name="_namespace" type="xs:string"/>
<xs:attribute name="_inherit-namespaces" type="xs:string"/>
<xs:attribute name="_use-attribute-sets" type="xs:string"/>
<xs:attribute name="_type" type="xs:string"/>
<xs:attribute name="_validation" type="xs:string"/>
<xs:assert test="exists(@name | @_name)" />
<xs:assert test="not(exists(@type | @_type) and exists(@validation | @_validation))">
    <xs:annotation>
        <xs:documentation>
            <p>
                The type and validation attributes are mutually exclusive (if
                one is present, the other must be absent).
            </p>
        </xs:documentation>
    </xs:annotation>
</xs:assert>
</xs:extension>
</xs:complexType>
</xs:element>

<xs:element name="evaluate" substitutionGroup="xsl:instruction">
<xs:complexType>
    <xs:complexContent mixed="true">
        <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:attribute name="xpath" type="xsl:expression"/>
            <xs:attribute name="as" type="xsl:sequence-type"/>
            <xs:attribute name="base-uri" type="xsl:avt"/>
            <xs:attribute name="context-item" type="xsl:expression"/>
            <xs:attribute name="namespace-context" type="xsl:expression"/>
            <xs:attribute name="schema-aware" type="xsl:avt"/>
            <xs:attribute name="with-params" type="xsl:expression"/>
            <xs:attribute name="_xpath" type="xs:string"/>
            <xs:attribute name="_as" type="xs:string"/>
            <xs:attribute name="_base-uri" type="xs:string"/>
            <xs:attribute name="_context-item" type="xs:string"/>
            <xs:attribute name="_namespace-context" type="xs:string"/>
            <xs:attribute name="_schema-aware" type="xs:string"/>
            <xs:attribute name="_with-params" type="xs:string"/>
            <xs:assert test="exists(@xpath | @_xpath)" />
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
</xs:element>

<xs:element name="expose">
<xs:annotation>
    <xs:documentation>
        <p>
            This element appears as a child of xsl:use-package and defines the
            visibility of components that are made available (or not) by this
            package to other using packages.
        </p>
    </xs:documentation>

```

```

</p>
</xs:documentation>
</xs:annotation>
<xs:complexType>
  <xs:complexContent>
    <xs:extension base="xsl:element-only-versioned-element-type">
      <xs:attribute name="component" type="xsl:component-kind-type"/>
      <xs:attribute name="names" type="xsl:EQNames"/>
      <xs:attribute name="visibility" type="xsl:visibility-not-hidden-type"/>
      <xs:attribute name="_component" type="xs:string"/>
      <xs:attribute name="_names" type="xs:string"/>
      <xs:attribute name="_visibility" type="xs:string"/>
    </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"/>
        <xs:attribute name="_select" type="xs:string"/>
        <xs:assert test="every $e in subsequence(xsl:sort, 2)
                      satisfies empty($e/(@stable | @_stable))">
          <xs:annotation>
            <xs:documentation>
              <p>
                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.
              </p>
            </xs:documentation>
          </xs:annotation>
        </xs:assert>
        <xs:assert test="exists(@select | @_select)" />
      </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"/>
<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="composite" type="xsl:yes-or-no"/>
<xs:attribute name="collation" type="xsl:avt"/>
<xs:attribute name="_select" type="xs:string"/>
<xs:attribute name="_group-by" type="xs:string"/>
<xs:attribute name="_group-adjacent" type="xs:string"/>
<xs:attribute name="_group-starting-with" type="xs:string"/>
<xs:attribute name="_group-ending-with" type="xs:string"/>
<xs:attribute name="_composite" type="xs:string"/>
<xs:attribute name="_collation" type="xs:string"/>
<xs:assert test="exists(@select | @_select)"/>
<xs:assert test="every $e in subsequence(xsl:sort, 2)
                  satisfies empty($e/(@stable | @_stable))">
<xs:annotation>
  <xs:documentation>
    <p>
      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.
    </p>
  </xs:documentation>
</xs:annotation>
</xs:assert>
<xs:assert test="count(((@group-by| @_group-by)[1],
                      (@group-adjacent| @_group-adjacent)[1],
                      (@group-starting-with| @_group-starting-with)[1],
                      (@group-ending-with| @_group-ending-with)[1])) = 1">
<xs:annotation>
  <xs:documentation>
    <p>
      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.
    </p>
  </xs:documentation>
</xs:annotation>
</xs:assert>
<xs:assert test="if (exists(@collation| @_collation) or exists(@composite| @_composite)
                  then (exists(@group-by| @_group-by) or exists(@group-adjacent| @_group
                  else true())">
<xs:annotation>
  <xs:documentation>
    <p>
      It is an error to specify the collation attribute or the
      composite attribute if neither the group-by attribute nor
      group-adjacent attribute is specified.
    </p>
  </xs:documentation>
</xs:annotation>
</xs:assert>
</xs:extension>
</xs:complexType>
</xs:complexType>
</xs:element>

<xs:element name="fork" substitutionGroup="xsl:instruction">
  <xs:complexType>

```

```

<xs:complexType mixed="true">
  <xs:extension base="xsl:versioned-element-type">
    <xs:sequence>
      <xs:element ref="xsl:fallback" minOccurs="0" maxOccurs="unbounded"/>
      <xs:choice>
        <xs:sequence minOccurs="0" maxOccurs="unbounded">
          <xs:element ref="xsl:sequence"/>
          <xs:element ref="xsl:fallback" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:sequence>
          <xs:element ref="xsl:for-each-group"/>
          <xs:element ref="xsl:fallback" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:choice>
    </xs:sequence>
  </xs:extension>
</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-in-namespace"/>
        <xs:attribute name="override" type="xsl:yes-or-no" default="yes"/>
        <xs:attribute name="as" type="xsl:sequence-type" default="item()*/>
        <xs:attribute name="visibility" type="xsl:visibility-type"/>
        <xs:attribute name="streamability" type="xsl:streamability-type"/>
        <xs:attribute name="override-extension-function" type="xsl:yes-or-no"/>
        <xs:attribute name="new-each-time" type="xsl:yes-or-no-or-maybe"/>
        <xs:attribute name="cache" type="xsl:yes-or-no"/>
        <xs:attribute name="_name" type="xs:string"/>
        <xs:attribute name="_override" type="xs:string"/>
        <xs:attribute name="_as" type="xs:string"/>
        <xs:attribute name="_visibility" type="xs:string"/>
        <xs:attribute name="_streamability" type="xs:string"/>
        <xs:attribute name="_override-extension-function" type="xs:string"/>
        <xs:attribute name="_identity-sensitive" type="xs:string"/>
        <xs:attribute name="_cache" type="xs:string"/>
        <xs:assert test="exists(@name | @_name)" />
        <xs:assert test="every $e in xsl:param
                      satisfies (empty($e/(@select | @_select)) and empty($e/child::node()))
                      <xs:annotation>
                        <xs:documentation>
                          <p>
                            A parameter for a function must have no default value.
                          </p>
                        </xs:documentation>
                      </xs:annotation>
                    </xs:assert>
                    <xs:assert test="every $e in xsl:param satisfies empty($e/(@visibility | @_visibility
                      <xs:annotation>
                        <xs:documentation>
                          <p>
```

```

        A parameter for a function must have no visibility attribute.
    </p>
    </xs:documentation>
    </xs:annotation>
</xs:assert>
<xs:assert test="every $e in xsl:param satisfies empty($e/(@required | @_required))">
    <xs:annotation>
        <xs:documentation>
            <p>
                A parameter for a function must have no required attribute.
            </p>
        </xs:documentation>
    </xs:annotation>
</xs:assert>
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>

<xs:element name="global-context-item" substitutionGroup="xsl:declaration">
<xs:complexType>
    <xs:complexContent>
        <xs:extension base="xsl:element-only-versioned-element-type">
            <xs:attribute name="as" type="xsl:item-type"/>
            <xs:attribute name="use">
                <xs:simpleType>
                    <xs:restriction base="xs:token">
                        <xs:enumeration value="required"/>
                        <xs:enumeration value="optional"/>
                        <xs:enumeration value="absent"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:attribute>
            <xs:attribute name="_as" type="xs:string"/>
            <xs:attribute name="_use" type="xs:string"/>
        </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"/>
            <xs:attribute name="_test" type="xs:string"/>
            <xs:assert test="exists(@test | @_test)"/>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
</xs:element>

<xs:element name="import" substitutionGroup="xsl:declaration">
<xs:complexType>
    <xs:complexContent>
        <xs:extension base="xsl:element-only-versioned-element-type">
            <xs:attribute name="href" type="xs:anyURI"/>
            <xs:attribute name="_href" type="xs:string"/>
            <xs:assert test="exists(@href | @_href)"/>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>

```

```

</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:attribute name="_namespace" type="xs:string"/>
        <xs:attribute name="_schema-location" type="xs:string"/>
        <xs:assert test="not(exists(@schema-location | @_schema-location) and exists(xs:schem
          <xs:annotation>
            <xs:documentation>
              <p>
                XTSE0215: It is a static error if an xsl:import-schema element
                that contains an xs:schema element has a schema-location
                attribute
              </p>
            </xs:documentation>
          </xs:annotation>
        </xs:assert>
      </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"/>
        <xs:attribute name="_href" type="xs:string"/>
        <xs:assert test="exists(@href | @_href)"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="iterate" substitutionGroup="xsl:instruction">
  <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:element ref="xsl:on-completion" minOccurs="0" maxOccurs="1"/>
          <xs:group ref="xsl:sequence-constructor-group"
            minOccurs="0"
            maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="select" type="xsl:expression"/>
        <xs:attribute name="_select" type="xs:string"/>
        <xs:assert test="exists(@select | @_select)"/>
      </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"/>
        <xs:attribute name="match" type="xsl:pattern"/>
        <xs:attribute name="use" type="xsl:expression"/>
        <xs:attribute name="composite" type="xsl:yes-or-no"/>
        <xs:attribute name="collation" type="xs:anyURI"/>
        <xs:attribute name="_name" type="xs:string"/>
        <xs:attribute name="_match" type="xs:string"/>
        <xs:attribute name="_use" type="xs:string"/>
        <xs:attribute name="_composite" type="xs:string"/>
        <xs:attribute name="_collation" type="xs:string"/>
        <xs:assert test="exists(@name | @_name)" />
        <xs:assert test="exists(@match | @_match)" />
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="map"
  type="xsl:sequence-constructor"
  substitutionGroup="xsl:instruction"/>

<xs:element name="map-entry" substitutionGroup="xsl:instruction">
  <xs:complexType>
    <xs:complexContent mixed="true">
      <xs:extension base="xsl:sequence-constructor-and-select">
        <xs:attribute name="key" type="xsl:expression"/>
        <xs:attribute name="_key" type="xs:string"/>
        <xs:assert test="exists(@key | @_key)" />
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="matching-substring" type="xsl:sequence-constructor"/>

<xs:element name="merge" substitutionGroup="xsl:instruction">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="xsl:element-only-versioned-element-type">
        <xs:sequence>
          <xs:element ref="xsl:merge-source" minOccurs="1" maxOccurs="unbounded"/>
          <xs:element ref="xsl:merge-action" minOccurs="1" maxOccurs="1"/>
          <xs:element ref="xsl:fallback" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="merge-action" type="xsl:sequence-constructor"/>

<xs:element name="merge-key" substitutionGroup="xsl:instruction">
  <xs:complexType>
    <xs:complexContent mixed="true">
      <xs:extension base="xsl:versioned-element-type">
        <xs:sequence>

```

```

<xs:group ref="xsl:sequence-constructor-group"
           minOccurs="0"
           maxOccurs="unbounded"/>
</xs:sequence>
<xs:attribute name="select" type="xsl:expression"/>
<xs:attribute name="lang" type="xsl:avt"/>
<xs:attribute name="order" type="xsl:avt"/>
<xs:attribute name="collation" type="xs:anyURI"/>
<xs:attribute name="case-order" type="xsl:avt"/>
<xs:attribute name="data-type" type="xsl:avt"/>
<xs:attribute name="_select" type="xs:string"/>
<xs:attribute name="_lang" type="xs:string"/>
<xs:attribute name="_order" type="xs:string"/>
<xs:attribute name="_collation" type="xs:string"/>
<xs:attribute name="_case-order" type="xs:string"/>
<xs:attribute name="_data-type" type="xs:string"/>
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>

<xs:element name="merge-source">
<xs:complexType>
<xs:complexContent>
<xs:extension base="xsl:element-only-versioned-element-type">
<xs:sequence>
<xs:element ref="xsl:merge-key" minOccurs="1" maxOccurs="unbounded"/>
</xs:sequence>
<xs:attribute name="name" type="xs:NCName"/>
<xs:attribute name="for-each-item" type="xsl:expression"/>
<xs:attribute name="for-each-source" type="xsl:expression"/>
<xs:attribute name="select" type="xsl:expression"/>
<xs:attribute name="streamable" type="xsl:yes-or-no"/>
<xs:attribute name="use-accumulators" type="xsl:accumulator-names"/>
<xs:attribute name="sort-before-merge" type="xsl:yes-or-no"/>
<xs:attribute name="type" type="xsl:EQName"/>
<xs:attribute name="validation" type="xsl:validation-type"/>
<xs:attribute name="_name" type="xs:string"/>
<xs:attribute name="_for-each-item" type="xs:string"/>
<xs:attribute name="_for-each-source" type="xs:string"/>
<xs:attribute name="_select" type="xs:string"/>
<xs:attribute name="_streamable" type="xs:string"/>
<xs:attribute name="_use-accumulators" type="xs:string"/>
<xs:attribute name="_sort-before-merge" type="xs:string"/>
<xs:attribute name="_type" type="xs:string"/>
<xs:attribute name="_validation" type="xs:string"/>
<xs:assert test="exists(@select | @_select)"/>
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>

<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:attribute name="error-code" type="xsl:avt"/>
<xs:attribute name="_select" type="xs:string"/>
<xs:attribute name="_terminate" type="xs:string"/>

```

```

        <xs:attribute name="_error-code" type="xs:string"/>
    </xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>

<xs:element name="mode" substitutionGroup="xsl:declaration">
<xs:complexType>
<xs:complexContent mixed="false">
    <xs:extension base="xsl:element-only-versioned-element-type">
        <xs:attribute name="name" type="xsl:QName"/>
        <xs:attribute name="streamable" type="xsl:yes-or-no" default="no"/>
        <xs:attribute name="use-accumulators" type="xsl:accumulator-names"/>
        <xs:attribute name="on-no-match" type="xsl:on-no-match-type" default="shallow-skip"/>
        <xs:attribute name="on-multiple-match"
                      type="xsl:on-multiple-match-type"
                      default="use-last"/>
        <xs:attribute name="warning-on-no-match" type="xsl:yes-or-no"/>
        <xs:attribute name="warning-on-multiple-match" type="xsl:yes-or-no"/>
        <xs:attribute name="typed" type="xsl:typed-type"/>
        <xs:attribute name="visibility">
            <xs:simpleType>
                <xs:restriction base="xsl:visibility-type">
                    <xs:enumeration value="public"/>
                    <xs:enumeration value="private"/>
                    <xs:enumeration value="final"/>
                </xs:restriction>
            </xs:simpleType>
        </xs:attribute>
        <xs:attribute name="_name" type="xs:string"/>
        <xs:attribute name="_streamable" type="xs:string"/>
        <xs:attribute name="_on-no-match" type="xs:string"/>
        <xs:attribute name="_on-multiple-match" type="xs:string"/>
        <xs:attribute name="_warning-on-no-match" type="xs:string"/>
        <xs:attribute name="_warning-on-multiple-match" type="xs:string"/>
        <xs:attribute name="_typed" type="xs:string"/>
        <xs:attribute name="_visibility" type="xs:string"/>
    </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-or-select">
        <xs:attribute name="name" type="xsl:avt"/>
        <xs:attribute name="_name" type="xs:string"/>
        <xs:assert test="exists(@name | @_name)"/>
    </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"/>
        <xs:attribute name="result-prefix" type="xsl:prefix-or-default"/>

```

```

<xs:attribute name="_stylesheet-prefix" type="xs:string"/>
<xs:attribute name="_result-prefix" type="xs:string"/>
<xs:assert test="exists(@stylesheet-prefix | @_stylesheet-prefix)" />
<xs:assert test="exists(@result-prefix | @_result-prefix)" />
<xs:assert test="every $prefix in (@stylesheet-prefix, @result-prefix)
                  /normalize-space(.)[. ne '#default']
                  satisfies $prefix = in-scope-prefixes(.)" />
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>

<xs:element name="next-iteration" substitutionGroup="xsl:instruction">
<xs:complexType>
<xs:complexContent mixed="true">
<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="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="start-at" type="xsl:avt"/>
<xs:attribute name="grouping-separator" type="xsl:avt"/>
<xs:attribute name="grouping-size" type="xsl:avt"/>
<xs:attribute name="_value" type="xs:string"/>
<xs:attribute name="_select" type="xs:string"/>
<xs:attribute name="_level" type="xs:string"/>
<xs:attribute name="_count" type="xs:string"/>

```

```

<xs:attribute name="_from" type="xs:string"/>
<xs:attribute name="_format" type="xs:string"/>
<xs:attribute name="_lang" type="xs:string"/>
<xs:attribute name="_letter-value" type="xs:string"/>
<xs:attribute name="_ordinal" type="xs:string"/>
<xs:attribute name="_start-at" type="xs:string"/>
<xs:attribute name="_grouping-separator" type="xs:string"/>
<xs:attribute name="_grouping-size" type="xs:string"/>
<xs:assert test="if (exists(@value | @_value))
    then empty(@select | @_select, @count | @_count, @from | @_from)
    and (exists(@_level) or normalize-space(@level)='single')
    else true()">
<xs:annotation>
  <xs:documentation>
    <p>
      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.
    </p>
  </xs:documentation>
</xs:annotation>
</xs:assert>
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>

<xs:element name="on-completion" type="xsl:sequence-constructor-or-select"/>

<xs:element name="on-empty"
  substitutionGroup="xsl:instruction"
  type="xsl:sequence-constructor-or-select"/>

<xs:element name="on-non-empty"
  substitutionGroup="xsl:instruction"
  type="xsl:sequence-constructor-or-select"/>

<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="allow-duplicate-names" type="xsl:yes-or-no"/>
        <xs:attribute name="build-tree" type="xsl:yes-or-no"/>
        <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="html-version" type="xs:decimal"/>
        <xs:attribute name="include-content-type" type="xsl:yes-or-no"/>
        <xs:attribute name="indent" type="xsl:yes-or-no"/>
        <xs:attribute name="item-separator" type="xs:string"/>
        <xs:attribute name="json-node-output-method" type="xsl:method"/>
        <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:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

```

```

<xs:attribute name="parameter-document" type="xs:anyURI"/>
<xs:attribute name="standalone" type="xsl:yes-or-no-or-omit"/>
<xs:attribute name="suppress-indentation" type="xsl:EQNames"/>
<xs:attribute name="undeclare-prefixes" type="xsl:yes-or-no"/>
<xs:attribute name="use-character-maps" type="xsl:EQNames"/>
<xs:attribute name="version" type="xs:NMTOKEN"/>
<xs:attribute name="_name" type="xs:string"/>
<xs:attribute name="_method" type="xs:string"/>
<xs:attribute name="_byte-order-mark" type="xs:string"/>
<xs:attribute name="_cdata-section-elements" type="xs:string"/>
<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="xs:string"/>
<xs:attribute name="_html-version" type="xs:string"/>
<xs:attribute name="_include-content-type" type="xs:string"/>
<xs:attribute name="_indent" type="xs:string"/>
<xs:attribute name="_item-separator" type="xs:string"/>
<xs:attribute name="_media-type" type="xs:string"/>
<xs:attribute name="_normalization-form" type="xs:string"/>
<xs:attribute name="_omit-xml-declaration" type="xs:string"/>
<xs:attribute name="_parameter-document" type="xs:string"/>
<xs:attribute name="_standalone" type="xs:string"/>
<xs:attribute name="_suppress-indentation" type="xs:string"/>
<xs:attribute name="_undeclare-prefixes" type="xs:string"/>
<xs:attribute name="_use-character-maps" type="xs:string"/>
<xs:attribute name="_version" type="xs:string"/>
</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"/>
<xs:attribute name="string" type="xs:string"/>
<xs:attribute name="_character" type="xs:string"/>
<xs:attribute name="_string" type="xs:string"/>
<xs:assert test="exists(@character | @_character)"/>
<xs:assert test="exists(@string | @_string)"/>
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>

<xs:element name="override">
<xs:annotation>
<xs:documentation>
<p>
This element appears as a child of xsl:use-package and defines any
overriding definitions of components that the containing package wishes
to make to the components made available from a library package.
</p>
</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:complexContent>
<xs:extension base="xsl:element-only-versioned-element-type">
<xs:choice minOccurs="0" maxOccurs="unbounded">

```

```

<xs:element ref="xsl:template"/>
<xs:element ref="xsl:function"/>
<xs:element ref="xsl:variable"/>
<xs:element ref="xsl:param"/>
<xs:element ref="xsl:attribute-set"/>
</xs:choice>
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>

<xs:element name="package">
<xs:complexType>
<xs:complexContent>
<xs:extension base="xsl:element-only-versioned-element-type">
<xs:sequence>
<xs:choice minOccurs="0" maxOccurs="unbounded">
<xs:element ref="xsl:expose"/>
<xs:element ref="xsl:declaration"/>
<xs:any namespace="#other" processContents="lax"/>
</xs:choice>
</xs:sequence>
<xs:attribute name="declared-modes" type="xsl:yes-or-no"/>
<xs:attribute name="id" type="xs:ID"/>
<xs:attribute name="name" type="xs:anyURI"/>
<xs:attribute name="package-version" type="xs:string"/>
<xs:attribute name="input-type-annotations" type="xsl:input-type-annotations-type"/>
<xs:attribute name="_declared-modes" type="xs:string"/>
<xs:attribute name="_id" type="xs:string"/>
<xs:attribute name="_name" type="xs:string"/>
<xs:attribute name="_package-version" type="xs:string"/>
<xs:attribute name="_input-type-annotations" type="xs:string"/>
</xs:sequence>
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>

<xs:element name="param" substitutionGroup="xsl:declaration">
<xs:annotation>
<xs:documentation>
<p>
Declaration of the xsl:param element, used both defining function
parameters, template parameters, parameters to xsl:iterate, and global
stylesheet parameters.
</p>
</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:complexContent mixed="true">
<xs:extension base="xsl:sequence-constructor-or-select">
<xs:attribute name="name" type="xsl:QName"/>
<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:attribute name="static" type="xsl:yes-or-no"/>
<xs:attribute name="_name" type="xs:string"/>
<xs:attribute name="_as" type="xs:string"/>
<xs:attribute name="_required" type="xs:string"/>
<xs:attribute name="_tunnel" type="xs:string"/>
<xs:attribute name="_static" type="xs:string"/>
<xs:assert test="exists(@name | @_name)"/>

```

```

<xs:assert test="if (normalize-space(@static) = ('yes', 'true', '1'))
    then empty(*,text())
    else true()">
    <xs:annotation>
        <xs:documentation>
            <p>
                When the attribute static="yes" is specified, the xsl:param
                element must have empty content.
            </p>
        </xs:documentation>
    </xs:annotation>
</xs:assert>
</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:assert test="every $e in subsequence(xsl:sort, 2)
                    satisfies empty($e/(@stable | @_stable))">
                    <xs:annotation>
                        <xs:documentation>
                            <p>
                                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.
                            </p>
                        </xs:documentation>
                    </xs:annotation>
                </xs:assert>
            </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"/>
                <xs:attribute name="_elements" type="xs:string"/>
                <xs:assert test="exists(@elements | @_elements)">
                </xs:extension>
            </xs:complexContent>
        </xs:complexType>
</xs:element>

<xs:element name="processing-instruction" substitutionGroup="xsl:instruction">
    <xs:complexType>

```

```

<xs:complexType mixed="true">
  <xs:extension base="xsl:sequence-constructor-or-select">
    <xs:attribute name="name" type="xsl:avt"/>
    <xs:attribute name="_name" type="xs:string"/>
    <xs:assert test="exists(@name | @_name)" />
  </xs:extension>
</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:EQName"/>
        <xs:attribute name="validation" type="xsl:validation-type"/>
        <xs:attribute name="method" type="xsl:avt"/>
        <xs:attribute name="allow-duplicate-names" type="xsl:avt"/>
        <xs:attribute name="build-tree" 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="html-version" type="xsl:avt"/>
        <xs:attribute name="include-content-type" type="xsl:avt"/>
        <xs:attribute name="indent" type="xsl:avt"/>
        <xs:attribute name="item-separator" type="xsl:avt"/>
        <xs:attribute name="json-node-output-method" 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="parameter-document" type="xsl:avt"/>
        <xs:attribute name="standalone" type="xsl:avt"/>
        <xs:attribute name="suppress-indentation" type="xsl:avt"/>
        <xs:attribute name="undeclare-prefixes" type="xsl:avt"/>
        <xs:attribute name="use-character-maps" type="xsl:EQNames"/>
        <xs:attribute name="output-version" type="xsl:avt"/>
        <xs:attribute name="_format" type="xs:string"/>
        <xs:attribute name="_href" type="xs:string"/>
        <xs:attribute name="_type" type="xs:string"/>
        <xs:attribute name="_validation" type="xs:string"/>
        <xs:attribute name="_method" type="xs:string"/>
        <xs:attribute name="_byte-order-mark" type="xs:string"/>
        <xs:attribute name="_cdata-section-elements" type="xs:string"/>
        <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="xs:string"/>
        <xs:attribute name="_html-version" type="xs:string"/>
        <xs:attribute name="_include-content-type" type="xs:string"/>
        <xs:attribute name="_indent" type="xs:string"/>
        <xs:attribute name="_item-separator" type="xs:string"/>
        <xs:attribute name="_media-type" type="xs:string"/>
        <xs:attribute name="_normalization-form" type="xs:string"/>
        <xs:attribute name="_omit-xml-declaration" type="xs:string"/>
        <xs:attribute name="_parameter-document" type="xs:string"/>
        <xs:attribute name="_standalone" type="xs:string"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

```

```

<xs:attribute name="_suppress-indentation" type="xs:string"/>
<xs:attribute name="_undeclare-prefixes" type="xs:string"/>
<xs:attribute name="_use-character-maps" type="xs:string"/>
<xs:attribute name="_output-version" type="xs:string"/>
<xs:assert test="not(exists(@type | @_type) and exists(@validation | @_validation))">
    <xs:annotation>
        <xs:documentation>
            <p>
                The type and validation attributes are mutually exclusive (if
                one is present, the other must be absent).
            </p>
        </xs:documentation>
    </xs:annotation>
</xs:assert>
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>

<xs:element name="sequence"
            substitutionGroup="xsl:instruction"
            type="xsl:sequence-constructor-or-select"/>

<xs:element name="sort">
<xs:complexType>
    <xs:complexContent mixed="true">
        <xs:extension base="xsl:sequence-constructor-or-select">
            <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:avt"/>
            <xs:attribute name="_lang" type="xs:string"/>
            <xs:attribute name="_data-type" type="xs:string"/>
            <xs:attribute name="_order" type="xs:string"/>
            <xs:attribute name="_case-order" type="xs:string"/>
            <xs:attribute name="_collation" type="xs:string"/>
            <xs:attribute name="_stable" type="xs:string"/>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
</xs:element>

<xs:element name="source-document" substitutionGroup="xsl:instruction">
<xs:complexType>
    <xs:complexContent>
        <xs:extension base="xsl:sequence-constructor">
            <xs:attribute name="href" type="xsl:avt"/>
            <xs:attribute name="streamable" type="xsl:yes-or-no" default="no"/>
            <xs:attribute name="use-accumulators" type="xsl:accumulator-names"/>
            <xs:attribute name="type" type="xsl:QName"/>
            <xs:attribute name="validation" type="xsl:validation-type"/>
            <xs:attribute name="_href" type="xs:string"/>
            <xs:attribute name="_streamable" type="xs:string"/>
            <xs:attribute name="_use-accumulators" type="xs:string"/>
            <xs:attribute name="_type" type="xs:string"/>
            <xs:attribute name="_validation" type="xs:string"/>
            <xs:assert test="exists(@href | @_href)" />
            <xs:assert test="not(exists(@type | @_type) and exists(@validation | @_validation))">
                <xs:annotation>

```

```

<xs:documentation>
  <p>
    The type and validation attributes are mutually exclusive (if
    one is present, the other must be absent).
  </p>
</xs:documentation>
</xs:annotation>
</xs:assert>
</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"/>
        <xs:attribute name="_elements" type="xs:string"/>
        <xs:assert test="exists(@elements | @_elements)" />
      </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:context-item" minOccurs="0" maxOccurs="1"/>
          <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:attribute name="visibility" type="xsl:visibility-type"/>
        <xs:attribute name="_match" type="xs:string"/>
        <xs:attribute name="_priority" type="xs:string"/>
        <xs:attribute name="_mode" type="xs:string"/>
        <xs:attribute name="_name" type="xs:string"/>
        <xs:attribute name="_as" type="xs:string"/>
        <xs:attribute name="_visibility" type="xs:string"/>
        <xs:assert test="exists(@match | @_match) or exists(@name | @_name)" />
      <xs:annotation>
        <xs:documentation>
          <p>
            An xsl:template element must have either a match attribute or a
            name attribute, or both.
          </p>
        </xs:documentation>
      </xs:annotation>
    </xs:assert>
  </xs:element>

```

```

<xsl:assert test="if (empty(@match | @_match))
                  then (empty(@mode | @_mode) and empty(@priority | @_priority))
                  else true()">
  <xsl:annotation>
    <xsl:documentation>
      <p>
        An xsl:template element that has no match attribute must have no
        mode attribute and no priority attribute.
      </p>
    </xsl:documentation>
  </xsl:annotation>
</xsl:assert>
<xsl:assert test="not(exists(@visibility | @_visibility) and empty(@name | @_name))">
  <xsl:annotation>
    <xsl:documentation>
      <p>
        An xsl:template element that has no name attribute must have no
        visibility attribute
      </p>
    </xsl:documentation>
  </xsl:annotation>
</xsl:assert>
<xsl:assert test="if (normalize-space(@visibility) = 'abstract')
                  then empty(* except (xsl:context-item, xsl:param))
                  else true()">
  <xsl:annotation>
    <xsl:documentation>
      <p>
        If the visibility attribute is present with the value abstract
        then (a) the sequence constructor defining the template body
        must be empty: that is, the only permitted children are
        xsl:context-item and xsl:param
      </p>
    </xsl:documentation>
  </xsl:annotation>
</xsl:assert>
<xsl:assert test="not(normalize-space(@visibility) = 'abstract' and exists(@match))">
  <xsl:annotation>
    <xsl:documentation>
      <p>
        If the visibility attribute is present with the value abstract
        then there must be no match attribute.
      </p>
    </xsl:documentation>
  </xsl:annotation>
</xsl:assert>
<xsl:assert test="every $e in xsl:param satisfies empty($e/(@visibility | @_visibility
  <xsl:annotation>
    <xsl:documentation>
      <p>
        A parameter for a template must have no visibility attribute.
      </p>
    </xsl:documentation>
  </xsl:annotation>
  </xsl:assert>
</xsl:extension>
</xsl:complexType>
</xsl:element>

<xsl: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:complexType name="text-element-type">
  <xs:simpleContent>
    <xs:extension base="xsl:text-element-base-type">
      <xs:attribute name="disable-output-escaping" type="xsl:yes-or-no" default="no"/>
      <xs:attribute name="_disable-output-escaping" type="xs:string"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>

<xs:element name="text"
  substitutionGroup="xsl:instruction"
  type="xsl:text-element-type"/>

<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="optional"/>
      <xs:attribute name="_version" type="xs:string">
        <xs:annotation>
          <xs:documentation>
            <p>
              The version attribute indicates the version of XSLT that the
              stylesheet module requires. The attribute is required, unless the
              xsl:stylesheet element is a child of an xsl:package element, in
              which case it is optional: the default is then taken from the
              parent xsl:package element.
            </p>
          </xs:documentation>
        </xs:annotation>
      </xs:attribute>
      <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:choice minOccurs="0" maxOccurs="unbounded">
            <xs:element ref="xsl:declaration"/>
            <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="input-type-annotations"
          type="xsl:input-type-annotations-type"
          default="unspecified"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

```

```

<xs:attribute name="_id" type="xs:string"/>
<xs:attribute name="_input-type-annotations" type="xs:string"/>
<!--* The 'static' attribute may be used on 'param' and 'variable'
     * only when they are top-level elements. *-->
<xs:assert test="every $v in (./xsl:param, ./xsl:variable)[exists(@static | @_stati
                     satisfies $v[parent::xsl:stylesheet or parent::xsl:transform or pare
<xs:annotation>
    <xs:documentation>
        <p>
            The static attribute must not be present on an xsl:variable or
            xsl:param element unless it is a top-level element.
        </p>
    </xs:documentation>
</xs:annotation>
</xs:assert>
<xs:assert test="every $prefix in (@exclude-result-prefixes[not(. = '#all')],"
                     @extension-element-prefixes)
                     satisfies ((if ($prefix = '#default') then '' else $prefix) = in-sco
<xs:annotation>
    <xs:documentation>
        <p>
            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.
        </p>
        <p>
            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.
        </p>
    </xs:documentation>
</xs:annotation>
</xs:assert>
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>

<xs:element name="try" substitutionGroup="xsl:instruction">
<xs:complexType>
    <xs:complexContent mixed="true">
        <xs:extension base="xsl:versioned-element-type">
            <xs:sequence>
                <xs:group ref="xsl:sequence-constructor-group"
                           minOccurs="0"
                           maxOccurs="unbounded"/>
                <xs:element ref="xsl:catch" minOccurs="1" maxOccurs="1"/>
                <xs:choice minOccurs="0" maxOccurs="unbounded">
                    <xs:element ref="xsl:catch"/>
                    <xs:element ref="xsl:fallback"/>
                </xs:choice>
            </xs:sequence>
            <xs:attribute name="rollback-output" type="xsl:yes-or-no" default="yes"/>
            <xs:attribute name="select" type="xsl:expression" use="optional"/>
            <xs:attribute name="_rollback-output" type="xs:string"/>
            <xs:attribute name="_select" type="xs:string"/>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
</xs:element>

```

```

<xsl:element name="use-package" substitutionGroup="xsl:declaration">
  <xsl:annotation>
    <xsl:documentation>
      <p>
        This element appears as a child of xsl:package and defines a dependency
        of the containing package on another package, identified by URI in the
        name attribute. The package-version attribute indicates which version of
        the library package is required, or may indicate a range of versions.
      </p>
    </xsl:documentation>
  </xsl:annotation>
  <xsl:complexType>
    <xsl:complexContent mixed="false">
      <xsl:extension base="xsl:element-only-versioned-element-type">
        <xsl:choice minOccurs="0" maxOccurs="unbounded">
          <xsl:element ref="xsl:accept"/>
          <xsl:element ref="xsl:override"/>
        </xsl:choice>
        <xsl:attribute name="name" type="xs:anyURI"/>
        <xsl:attribute name="package-version" type="xs:string"/>
        <xsl:attribute name="_name" type="xs:string"/>
        <xsl:attribute name="_package-version" type="xs:string"/>
      </xsl:extension>
    </xsl:complexContent>
  </xsl:complexType>
</xsl:element>

<xsl:element name="value-of" substitutionGroup="xsl:instruction">
  <xsl:complexType>
    <xsl:complexContent mixed="true">
      <xsl:extension base="xsl:sequence-constructor-or-select">
        <xsl:attribute name="separator" type="xsl:avt"/>
        <xsl:attribute name="disable-output-escaping" type="xsl:yes-or-no" default="no"/>
        <xsl:attribute name="_separator" type="xs:string"/>
        <xsl:attribute name="_disable-output-escaping" type="xs:string"/>
      </xsl:extension>
    </xsl:complexContent>
  </xsl:complexType>
</xsl:element>

<xsl:element name="variable" substitutionGroup="xsl:declaration xsl:instruction">
  <xsl:annotation>
    <xsl:documentation>
      <p>
        Declaration of the xsl:variable element, used both for local and global
        variable bindings.
      </p>
      <p>
        This definition takes advantage of the ability in XSD 1.1 for an element
        to belong to more than one substitution group. A global variable is a
        declaration, while a local variable can appear as an instruction in a
        sequence constructor.
      </p>
    </xsl:documentation>
  </xsl:annotation>
  <xsl:complexType>
    <xsl:complexContent mixed="true">
      <xsl:extension base="xsl:sequence-constructor-or-select">
        <xsl:attribute name="name" type="xsl:QName"/>
        <xsl:attribute name="as" type="xsl:sequence-type"/>
      </xsl:extension>
    </xsl:complexContent>
  </xsl:complexType>
</xsl:element>

```

```

<xs:attribute name="visibility" type="xsl:visibility-type"/>
<xs:attribute name="static" type="xsl:yes-or-no"/>
<xs:attribute name="_name" type="xs:string"/>
<xs:attribute name="_as" type="xs:string"/>
<xs:attribute name="_visibility" type="xs:string"/>
<xs:attribute name="_static" type="xs:string"/>
<xs:assert test="exists(@name | @_name)">
    <xs:assert test="if (normalize-space(@static) = ('yes', 'true', '1'))
        then (exists(@_visibility) or normalize-space(@visibility)
            = ('', 'private', 'final'))
        else true()">
        <xs:annotation>
            <xs:documentation>
                <p>
                    When the static attribute is present with the value yes, the
                    visibility attribute must not have a value other than private or
                    final.
                </p>
            </xs:documentation>
        </xs:annotation>
    </xs:assert>
    <xs:assert test="if (normalize-space(@static) = ('yes', 'true', '1'))
        then (empty(*, text()) and exists(@select | @_select))
        else true()">
        <xs:annotation>
            <xs:documentation>
                <p>
                    When the attribute static="yes" is specified, the xsl:variable
                    element must have empty content, and the select attribute must
                    be present to define the value of the variable.
                </p>
            </xs:documentation>
        </xs:annotation>
    </xs:assert>
    </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"/>
                <xs:attribute name="_test" type="xs:string"/>
                <xs:assert test="exists(@test | @_test)">
                </xs:assert>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>
</xs:element>

<xs:element name="where-populated"
    substitutionGroup="xsl:instruction"
    type="xsl:sequence-constructor"/>

<xs:element name="with-param">
    <xs:complexType>
        <xs:complexContent mixed="true">
            <xs:extension base="xsl:sequence-constructor-or-select">
                <xs:attribute name="name" type="xsl:QName"/>
                <xs:attribute name="as" type="xsl:sequence-type"/>

```

```

<xs:attribute name="tunnel" type="xsl:yes-or-no"/>
<xs:attribute name="_name" type="xs:string"/>
<xs:attribute name="_as" type="xs:string"/>
<xs:attribute name="_tunnel" type="xs:string"/>
<xs:assert test="exists(@name | @_name)" />
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>

<!-- ++++++ -->
<xs:annotation>
  <xs:documentation>
    <p>
      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.
    </p>
  </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="default-mode" type="xsl:default-mode-type"/>
  <xs:attribute name="default-validation"
    type="xsl:validation-strip-or-preserve"
    default="strip"/>
  <xs:attribute name="expand-text" type="xsl:yes-or-no"/>
  <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:EQNames"
    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:EQName"/>
  <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:group>

<!-- ++++++ -->
<xs:annotation>
  <xs:documentation>
    <p>
      PART D: definitions of simple types used in stylesheet attributes
    </p>
  </xs:documentation>
</xs:annotation>
<!-- ++++++ -->

<xs:simpleType name="accumulator-names">
  <xs:annotation>
    <xs:documentation>
      <p>
        The use-accumulators attribute of xsl:source-document, xsl:merge-source,
        or xsl:global-context-item: either a list, each member being a QName; or
        the value #all
      </p>
    </xs:documentation>
  </xs:annotation>
  <xs:union>
    <xs:simpleType>
      <xs:list itemType="xsl:QName"/>
    </xs:simpleType>
    <xs:simpleType>
      <xs:restriction base="xs:token">
        <xs:enumeration value="#all"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:union>
</xs:simpleType>

<xs:simpleType name="avt">
  <xs:annotation>
    <xs:documentation>
      <p>
        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 2.1 Recommendation.
      </p>
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:string"/>
</xs:simpleType>

<xs:simpleType name="char">
  <xs:annotation>
    <xs:documentation>
      <p>
        A string containing exactly one character.
      </p>
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:string">
    <xs:length value="1"/>
  </xs:restriction>
</xs:simpleType>

```

```

<xs:simpleType name="component-kind-type">
  <xs:annotation>
    <xs:documentation>
      <p>
        Describes a kind of component within a package.
      </p>
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:token">
    <xs:enumeration value="template"/>
    <xs:enumeration value="function"/>
    <xs:enumeration value="variable"/>
    <xs:enumeration value="attribute-set"/>
    <xs:enumeration value="mode"/>
    <xs:enumeration value="*"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="default-mode-type">
  <xs:annotation>
    <xs:documentation>
      <p>
        The default-mode attribute of xsl:stylesheet, xsl:transform, xsl:package
        (or any other xsl:* element): either a QName or #unnamed.
      </p>
    </xs:documentation>
  </xs:annotation>
  <xs:union memberTypes="xsl:EQName">
    <xs:simpleType>
      <xs:restriction base="xs:token">
        <xs:enumeration value="#unnamed"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:union>
</xs:simpleType>

<xs:simpleType name="expression">
  <xs:annotation>
    <xs:documentation>
      <p>
        An XPath 2.0 expression.
      </p>
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:token">
    <xs:pattern value=".+/">
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="item-type">
  <xs:annotation>
    <xs:documentation>
      <p>
        An XPath 2.1 ItemType
      </p>
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:token">
    <xs:pattern value=".+/">
  </xs:restriction>
</xs:simpleType>
```

```

</xs:simpleType>

<xs:simpleType name="input-type-annotations-type">
  <xs:annotation>
    <xs:documentation>
      <p>
        Describes how type annotations in source documents are handled.
      </p>
    </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>
      <p>
        The level attribute of xsl:number: one of single, multiple, or any.
      </p>
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:token">
    <xs:enumeration value="single"/>
    <xs:enumeration value="multiple"/>
    <xs:enumeration value="any"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="mode">
  <xs:annotation>
    <xs:documentation>
      <p>
        The mode attribute of xsl:apply-templates: either a QName, or #current,
        or #unnamed, or #default.
      </p>
    </xs:documentation>
  </xs:annotation>
  <xs:union memberTypes="xsl:EQName">
    <xs:simpleType>
      <xs:restriction base="xs:token">
        <xs:enumeration value="#default"/>
        <xs:enumeration value="#unnamed"/>
        <xs:enumeration value="#current"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:union>
</xs:simpleType>

<xs:simpleType name="modes">
  <xs:annotation>
    <xs:documentation>
      <p>
        The mode attribute of xsl:template: either a list, each member being
        either a QName or #default or #unnamed; or the value #all
      </p>
    </xs:documentation>
  </xs:annotation>
</xs:simpleType>

```

```

<xs:union>
  <xs:simpleType>
    <xs:restriction>
      <xs:simpleType>
        <xs:list>
          <xs:simpleType>
            <xs:union memberTypes="xsl:EQName">
              <xs:simpleType>
                <xs:restriction base="xs:token">
                  <xs:enumeration value="#default"/>
                  <xs:enumeration value="#unnamed"/>
                </xs:restriction>
              </xs:simpleType>
            </xs:union>
          </xs:simpleType>
        </xs:list>
      </xs:simpleType>
    </xs:restriction>
  </xs:simpleType>
<xs:assertion test="count($value) = count(distinct-values($value))">
  <xs:annotation>
    <xs:documentation>
      <p>
        XTSE0550: It is a static error if the same token is included
        more than once in the list.
      </p>
    </xs:documentation>
  </xs:annotation>
</xs:assertion>
</xs:restriction>
</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>
      <p>
        A list of NameTests, as defined in the XPath 2.0 Recommendation. Each
        NameTest is either a QName, or "*", or "prefix:*", or "*:localname"
      </p>
    </xs:documentation>
  </xs:annotation>
  <xs:list>
    <xs:simpleType>
      <xs:union memberTypes="xsl:EQName">
        <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:list>
  </xs:simpleType>

```

```

</xs:simpleType>
</xs:list>
</xs:simpleType>

<xs:simpleType name="on-multiple-match-type">
  <xs:annotation>
    <xs:documentation>
      <p>
        Describes the action to be taken when there are several template rules
        to match an item in a given mode.
      </p>
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:token">
    <xs:enumeration value="use-last"/>
    <xs:enumeration value="fail"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="on-no-match-type">
  <xs:annotation>
    <xs:documentation>
      <p>
        Describes the action to be taken when there is no template rule to match
        an item in a given mode.
      </p>
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:token">
    <xs:enumeration value="deep-copy"/>
    <xs:enumeration value="shallow-copy"/>
    <xs:enumeration value="deep-skip"/>
    <xs:enumeration value="shallow-skip"/>
    <xs:enumeration value="text-only-copy"/>
    <xs:enumeration value="fail"/>
  </xs:restriction>
</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>
      <p>
        The method attribute of xsl:output: Either one of the recognized names
      </p>
    </xs:documentation>
  </xs:annotation>
</xs:simpleType>

```

```

    "xml", "xhtml", "html", "text", or a QName that must include a prefix.
  </p>
  </xs:documentation>
</xs:annotation>
<xs:union>
  <xs:simpleType>
    <xs:restriction base="xs:token">
      <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>
      <p>
        A match pattern as defined in the XSLT 2.1 Recommendation. The syntax
        for patterns is a restricted form of the syntax for XPath 2.0
        expressions. Change since XSLT 2.0: Patterns may now match any item (not
        only nodes)
      </p>
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xsl:expression"/>
</xs:simpleType>

<xs:simpleType name="prefix-or-default">
  <xs:annotation>
    <xs:documentation>
      <p>
        Either a namespace prefix, or #default. Used in the xsl:namespace-alias
        element.
      </p>
    </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="EQNames">
  <xs:annotation>
    <xs:documentation>
      <p>
        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
      </p>
    </xs:documentation>
  </xs:annotation>

```

```

</xs:documentation>
</xs:annotation>
<xs:list itemType="xsl:EQName"/>
</xs:simpleType>

<xs:simpleType name="EQName">
  <xs:annotation>
    <xs:documentation>
      <p>
        An extended QName. This schema does not use the built-in type xs:QName,
        but rather defines its own QName type. This may be either a local name,
        or a prefixed QName, or a name written using the extended QName notation
        Q{uri}local
      </p>
      <p>
        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 datatype is
        therefore defined as a union of NCName and QName, so that an unprefixed
        name will be validated as an NCName and will therefore not be treated as
        having the semantics of an unprefixed xs:QName.
      </p>
    </xs:documentation>
  </xs:annotation>
  <xs:union memberTypes="xs:NCName xs:QName">
    <xs:simpleType>
      <xs:restriction base="xs:token">
        <xs:pattern value="Q\{\[^{}]*\}[\i-[:]][\c-[:]]*"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:union>
</xs:simpleType>

<xs:simpleType name="EQName-in-namespace">
  <xs:annotation>
    <xs:documentation>
      <p>
        A subtype of EQNames that excludes no-namespace names
      </p>
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xsl:EQName">
    <xs:pattern value="Q\{.\+\}.\+|\i\c*.\+"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="sequence-type">
  <xs:annotation>
    <xs:documentation>
      <p>
        The description of a datatype, conforming to the SequenceType production
        defined in the XPath 2.0 Recommendation
      </p>
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:token">
    <xs:pattern value=".+/">
  </xs:restriction>
</xs:simpleType>

```

```

<xs:simpleType name="streamability-type">
  <xs:annotation>
    <xs:documentation>
      <p>
        Describes the category to which a function belongs, with regards to its
        streaming behavior.
      </p>
    </xs:documentation>
  </xs:annotation>
  <xs:union memberTypes="xsl:EQName-in-namespace">
    <xs:simpleType>
      <xs:restriction base="xs:token">
        <xs:enumeration value="unclassified"/>
        <xs:enumeration value="absorbing"/>
        <xs:enumeration value="inspection"/>
        <xs:enumeration value="filter"/>
        <xs:enumeration value="shallow-descent"/>
        <xs:enumeration value="deep-descent"/>
        <xs:enumeration value="ascent"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:union>
</xs:simpleType>

<xs:simpleType name="typed-type">
  <xs:annotation>
    <xs:documentation>
      <p>
        Describes whether a mode is designed to match typed or untyped nodes.
      </p>
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:token">
    <xs:enumeration value="yes"/>
    <xs:enumeration value="no"/>
    <xs:enumeration value="true"/>
    <xs:enumeration value="false"/>
    <xs:enumeration value="1"/>
    <xs:enumeration value="0"/>
    <xs:enumeration value="strict"/>
    <xs:enumeration value="lax"/>
    <xs:enumeration value="unspecified"/>
  </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>
      <p>
        Describes different ways of type-annotating an element or attribute.
      </p>
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xsl:validation-type">
    <xs:enumeration value="preserve"/>
    <xs:enumeration value="strip"/>
  </xs:restriction>
</xs:simpleType>

```

```

</xs:restriction>
</xs:simpleType>

<xs:simpleType name="validation-type">
  <xs:annotation>
    <xs:documentation>
      <p>
        Describes different ways of type-annotating an element or attribute.
      </p>
    </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="visibility-type">
  <xs:annotation>
    <xs:documentation>
      <p>
        Describes the visibility of a component within a package.
      </p>
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:token">
    <xs:enumeration value="public"/>
    <xs:enumeration value="private"/>
    <xs:enumeration value="final"/>
    <xs:enumeration value="abstract"/>
    <xs:enumeration value="hidden"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="visibility-not-hidden-type">
  <xs:annotation>
    <xs:documentation>
      <p>
        Describes the visibility of a component within a package.
      </p>
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xsl:visibility-type">
    <xs:enumeration value="public"/>
    <xs:enumeration value="private"/>
    <xs:enumeration value="final"/>
    <xs:enumeration value="abstract"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="yes-or-no">
  <xs:annotation>
    <xs:documentation>
      <p>
        One of the values "yes" or "no": the values "true" or "false", or "1" or
        "0" are accepted as synonyms.
      </p>
    </xs:documentation>
  </xs:annotation>
</xs:simpleType>

```

```

<xs:restriction base="xs:token">
  <xs:enumeration value="yes"/>
  <xs:enumeration value="no"/>
  <xs:enumeration value="true"/>
  <xs:enumeration value="false"/>
  <xs:enumeration value="1"/>
  <xs:enumeration value="0"/>
</xs:restriction>
</xs:simpleType>

<xs:simpleType name="yes-or-no-or-maybe">
  <xs:annotation>
    <xs:documentation>
      <p>
        One of the values "yes" or "no" or "omit". The values "true" or "false",
        or "1" or "0" are accepted as synonyms of "yes" and "no" respectively.
      </p>
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:token">
    <xs:enumeration value="yes"/>
    <xs:enumeration value="no"/>
    <xs:enumeration value="true"/>
    <xs:enumeration value="false"/>
    <xs:enumeration value="1"/>
    <xs:enumeration value="0"/>
    <xs:enumeration value="maybe"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="yes-or-no-or-omit">
  <xs:annotation>
    <xs:documentation>
      <p>
        One of the values "yes" or "no" or "omit". The values "true" or "false",
        or "1" or "0" are accepted as synonyms of "yes" and "no" respectively.
      </p>
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:token">
    <xs:enumeration value="yes"/>
    <xs:enumeration value="no"/>
    <xs:enumeration value="true"/>
    <xs:enumeration value="false"/>
    <xs:enumeration value="1"/>
    <xs:enumeration value="0"/>
    <xs:enumeration value="omit"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="zero-digit">
  <xs:annotation>
    <xs:documentation>
      <p>
        A digit that has the numerical value zero.
      </p>
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xsl:char">
    <xs:pattern value="\p{Nd}"/>
    <xs:assertion test="matches(string-join(codepoints-to-string(

```

```
        for $i in 0 to 9 return string-to-codepoints($value) + $i), ''), '\p{Nd}{10}"/>
</xs:restriction>
</xs:simpleType>

</xs:schema>
```

H.2 Relax-NG Schema for XSLT Stylesheets

The following Relax-NG schema may be used to validate XSLT 3.0 stylesheet modules. Similar caveats apply as for the XSD 1.1 version.

A copy of this schema is available at [schema-for-xslt30.rnc](#)

```

# XSLT 3.0 Relax NG Schema
#
# Copyright (c) 2010-2016, Mohamed ZERGAOUI (Innovimax)
#
# All rights reserved.
#
# Redistribution and use in source and binary forms, with or without modification, are permitted
# Redistributions of source code must retain the above copyright notice, this list of condition
# Redistributions in binary form must reproduce the above copyright notice, this list of condit
# Neither the name of the Mohamed ZERGAOUI or Innovimax nor the names of its contributors may b
#
# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS O
# FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTO
# (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DAT
# STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE U
#
namespace local = ""
default namespace xsl = "http://www.w3.org/1999/XSL/Transform"
namespace xs = "http://www.w3.org/2001/XMLSchema"

start =
stylesheet.element
| transform.element
| package.element
| literal-result-element-as-stylesheet

sequence-constructor.model = (instruction.category | literal-result-element | text)*

literal-result-element-as-stylesheet =
element * - xsl:* {
    attribute xsl:version { decimal.datatype },
    literal-result-element-no-version.atts,
    sequence-constructor.model
}

literal-result-element =
element * - xsl:* {
    literal-result-element.atts,
    sequence-constructor.model
}

literal-result-element.atts =
literal-result-element-no-version.atts,
attribute xsl:version { text }?

# These attributes may also appear on a literal result element, but in this case, to distinguis
# the names of the attributes are in the XSLT namespace. They are thus typically written as
# xsl:default-collation,
# xsl:default-mode,
# xsl:default-validation,
# xsl:exclude-result-prefixes,
# xsl:expand-text,
# xsl:extension-element-prefixes,
# xsl:use-when,
# xsl:version,
# or xsl>xpath-default-namespace.
literal-result-element-no-version.atts =
    attribute * - xsl:* { avt.datatype }*
& attribute xsl:default-collation { uris.datatype }?
& attribute xsl:default-mode { eqname.datatype | '#unnamed' }?
& attribute xsl:default-validation { "preserve" | "strip" }?

```

```

& attribute xsl:exclude-result-prefixes { exclude.prefixes.datatype }? # or prefixes.datatype
& attribute xsl:expand-text { boolean.datatype }?
& attribute xsl:extension-element-prefixes { extension.prefixes.datatype }? # or prefixes.data
& attribute xsl:inherit-namespaces { boolean.datatype }?
& attribute xsl:on-empty { expression.datatype }?
& attribute xsl:use-attribute-sets { eqnames.datatype }?
& attribute xsl:use-when { expression.datatype }?
& attribute xsl>xpath-default-namespace { xsd:anyURI }?
& (attribute xsl:type { eqname.datatype }
| attribute xsl:validation { "strict" | "lax" | "preserve" | "strip" })?

top-level-extension =
element * - (xsl:* | local:*) {
    anyElement
}

anyElement =
grammar {
    start = any
    any =
        (attribute * { text }
        | text
        | element * { any })*
}
extension.atts = attribute * - (xsl:* | local:*) { text }*

declarations.model = (declaration.category | top-level-extension)*

# [Definition: There are a number of standard attributes that may appear on any XSLT element: s
# default-collation,
# default-mode,
# default-validation,
# exclude-result-prefixes,
# expand-text,
# extension-element-prefixes,
# use-when,
# version,
# and xpath-default-namespace.]]

global.atts =
    attribute default-collation { uris.datatype }?,
    attribute _default-collation { avt.datatype }?,
    attribute default-mode { eqname.datatype | '#unnamed' }?,
    attribute _default-mode { avt.datatype }?,
    attribute default-validation { "preserve" | "strip" }?,
    attribute _default-validation { avt.datatype }?,
    attribute exclude-result-prefixes { exclude.prefixes.datatype }?,
    attribute _exclude-result-prefixes { avt.datatype }?, # or prefixes.datatype ?
    attribute expand-text { boolean.datatype }?,
    attribute _expand-text { avt.datatype }?,
    attribute extension-element-prefixes { extension.prefixes.datatype }?,
    attribute _extension-element-prefixes { avt.datatype }?, # or prefixes.datatype ?
    attribute use-when { expression.datatype }?,
    attribute _use-when { avt.datatype }?,
    attribute version { decimal.datatype }?,
    attribute _version { avt.datatype }?,
    attribute xpath-default-namespace { uri.datatype }?,
    attribute _xpath-default-namespace { avt.datatype }?

```

```

global.atts.except.version =
  attribute default-collation { uris.datatype }?,
  attribute _default-collation { avt.datatype }?,
  attribute exclude-result-prefixes { exclude.prefixes.datatype }?,
  attribute _exclude-result-prefixes { avt.datatype }?, # or prefixes.datatype ?
  attribute expand-text { boolean.datatype }?,
  attribute _expand-text { avt.datatype }?,
  attribute extension-element-prefixes { extension.prefixes.datatype }?,
  attribute _extension-element-prefixes { avt.datatype }?, # or prefixes.datatype ?
  attribute use-when { expression.datatype }?,
  attribute _use-when { avt.datatype }?,
  attribute xpath-default-namespace { uri.datatype }?,
  attribute _xpath-default-namespace { avt.datatype }?

qname.datatype = xsd:QName
# Extract from XPath 3.0
#[94] QName ::= QName | URIQualifiedName
#[104] NCName ::= [http://www.w3.org/TR/REC-xml-names/#NT-QName]Names
#[105] NCName ::= [http://www.w3.org/TR/REC-xml-names/#NT-NCName]Names
#[99] URIQualifiedName ::= BracedURILiteral NCName
#[100] BracedURILiteral ::= "Q" "{" [^{}]* "}"

uri.qualified.name = xsd:token { pattern = "Q\{[^{\}]*\}[\i-[:]][\c-[:]]*" }
qname.strict = xsd:token { pattern = "[\i-[:]][\c-[:]]:[\i-[:]][\c-[:]]" }
eqname.datatype = xsd:QName | uri.qualified.name | qname.strict
qnames.datatype = list { qname.datatype* }
eqnames.datatype = list { eqname.datatype* }
ncname.datatype = xsd:NCName
prefix.datatype = xsd:NCName
boolean.datatype = "yes" | "no" | "true" | "false" | "0" | "1"
expression.datatype = text
char.datatype = xsd:string { length = "1" }
string.datatype = text
id.datatype = xsd:NCName
tokens.datatype = list { token* }
prefixes.datatype = list { token* }
extension.prefixes.datatype = list { xsd:NCName* }
exclude.prefixes.datatype = list { "#all" | (xsd:NCName | "#default")* }
token.datatype = token
language.datatype = xsd:language
nmtoken.datatype = xsd:NMTOKEN
decimal.datatype = xsd:decimal
integer.datatype = xsd:integer
uri.datatype = xsd:anyURI
uris.datatype = list { xsd:anyURI* }
pattern.datatype = text
qname-but-not-ncname.datatype = xsd:QName { pattern = ".*:.*" }
xs_schema.element = element xs:schema { anyElement* }
item-type.datatype = text
sequence-type.datatype = text

declaration.category =
  use-package.element
  | include.element
  | import.element
  | import-schema.element
  | strip-space.element
  | preserve-space.element

```

```
| decimal-format.element
| template.element
| mode.element
| global-context-item.element
| variable.element
| param.element
| attribute-set.element
| function.element
| namespace-alias.element
| accumulator.element
| key.element
| output.element
| character-map.element

instruction.category =
  apply-templates.element
| apply-imports.element
| next-match.element
| for-each.element
| iterate.element
| next-iteration.element
| break.element
| if.element
| choose.element
| try.element
| variable.element
| call-template.element
| evaluate.element
| element.element
| attribute.element
| text.element
| value-of.element
| document.element
| processing-instruction.element
| namespace.element
| comment.element
| copy.element
| copy-of.element
| sequence.element
| where-populated.element
| on-empty.element
| on-non-empty.element
| number.element
| perform-sort.element
| for-each-group.element
| merge.element
| fork.element
| analyze-string.element
| source-document.element
| map.element
| map-entry.element
| message.element
| assert.element
| fallback.element
| result-document.element
package.element =
  element package {
    extension.atts,
    attribute id { id.datatype }?,
    attribute _id { avt.datatype }?,
    attribute name { uri.datatype }?,
```

```

attribute _name { avt.datatype }?,
attribute package-version { string.datatype }?,
attribute _package-version { avt.datatype }?,
attribute version { decimal.datatype }?,
attribute _version { avt.datatype }?,
attribute input-type-annotations { "preserve" | "strip" | "unspecified" }?,
attribute _input-type-annotations { avt.datatype }?,
attribute declared-modes { boolean.datatype }?,
attribute _declared-modes { avt.datatype }?,
attribute default-mode { eqname.datatype | "#unnamed" }?,
attribute _default-mode { avt.datatype }?,
attribute default-validation { "preserve" | "strip" }?,
attribute _default-validation { avt.datatype }?,
attribute default-collation { uris.datatype }?,
attribute _default-collation { avt.datatype }?,
attribute extension-element-prefixes { prefixes.datatype }?,
attribute _extension-element-prefixes { avt.datatype }?,
attribute exclude-result-prefixes { prefixes.datatype }?,
attribute _exclude-result-prefixes { avt.datatype }?,
attribute expand-text { boolean.datatype }?,
attribute _expand-text { avt.datatype }?,
attribute use-when { expression.datatype }?,
attribute _use-when { avt.datatype }?,
attribute xpath-default-namespace { uri.datatype }?,
attribute _xpath-default-namespace { avt.datatype }?,
((expose.element | declarations.model)*)
}

use-package.element =
element use-package {
  extension.atts,
  global.atts,
  attribute name { uri.datatype }?,
  attribute _name { avt.datatype }?,
  attribute package-version { string.datatype }?,
  attribute _package-version { avt.datatype }?,
  (accept.element | override.element)*
}
expose.element =
element expose {
  extension.atts,
  global.atts,
  attribute component { "template" | "function" | "attribute-set" | "variable" | "mode" | "declare-variable-set" }?,
  attribute _component { avt.datatype }?,
  attribute names { tokens.datatype }?,
  attribute _names { avt.datatype }?,
  attribute visibility { "public" | "private" | "final" | "abstract" }?,
  attribute _visibility { avt.datatype }?,
  empty
}
accept.element =
element accept {
  extension.atts,
  global.atts,
  (attribute component { "template" | "function" | "attribute-set" | "variable" | "mode" | "declare-variable-set" }|,
  attribute _component { avt.datatype })|,
  (attribute names { tokens.datatype } |
  attribute _names { avt.datatype })|,
  (attribute visibility { "public" | "private" | "final" | "abstract" | "hidden" } |
  attribute _visibility { avt.datatype })|,
  empty
}

```

```

override.element =
  element override {
    extension.atts,
    global.atts,
    (template.element | function.element | variable.element | param.element | attribute-set.e
  }
stylesheet.element =
  element stylesheet {
    extension.atts,
    attribute id { id.datatype }?,
    attribute _id { avt.datatype }?,
    attribute version { decimal.datatype }?,
    attribute _version { avt.datatype }?,
    attribute default-mode { eqname.datatype | "#unnamed" }?,
    attribute _default-mode { avt.datatype }?,
    attribute default-validation { "preserve" | "strip" }?,
    attribute _default-validation { avt.datatype }?,
    attribute input-type-annotations { "preserve" | "strip" | "unspecified" }?,
    attribute _input-type-annotations { avt.datatype }?,
    attribute default-collation { uris.datatype }?,
    attribute _default-collation { avt.datatype }?,
    attribute extension-element-prefixes { prefixes.datatype }?,
    attribute _extension-element-prefixes { avt.datatype }?,
    attribute exclude-result-prefixes { prefixes.datatype }?,
    attribute _exclude-result-prefixes { avt.datatype }?,
    attribute expand-text { boolean.datatype }?,
    attribute _expand-text { avt.datatype }?,
    attribute use-when { expression.datatype }?,
    attribute _use-when { avt.datatype }?,
    attribute xpath-default-namespace { uri.datatype }?,
    attribute _xpath-default-namespace { avt.datatype }?,
    (declarations.model)
  }
transform.element =
  element transform {
    extension.atts,
    attribute id { id.datatype }?,
    attribute _id { avt.datatype }?,
    attribute version { decimal.datatype }?,
    attribute _version { avt.datatype }?,
    attribute default-mode { eqname.datatype | "#unnamed" }?,
    attribute _default-mode { avt.datatype }?,
    attribute default-validation { "preserve" | "strip" }?,
    attribute _default-validation { avt.datatype }?,
    attribute input-type-annotations { "preserve" | "strip" | "unspecified" }?,
    attribute _input-type-annotations { avt.datatype }?,
    attribute default-collation { uris.datatype }?,
    attribute _default-collation { avt.datatype }?,
    attribute extension-element-prefixes { prefixes.datatype }?,
    attribute _extension-element-prefixes { avt.datatype }?,
    attribute exclude-result-prefixes { prefixes.datatype }?,
    attribute _exclude-result-prefixes { avt.datatype }?,
    attribute expand-text { boolean.datatype }?,
    attribute _expand-text { avt.datatype }?,
    attribute use-when { expression.datatype }?,
    attribute _use-when { avt.datatype }?,
    attribute xpath-default-namespace { uri.datatype }?,
    attribute _xpath-default-namespace { avt.datatype }?,
    (declarations.model)
  }
include.element =

```

```

element include {
    extension.atts,
    global.atts,
    attribute href { uri.datatype }?,
    attribute _href { avt.datatype }?,
    empty
}
import.element =
    element import {
        extension.atts,
        global.atts,
        (attribute href { uri.datatype }
        | attribute _href { avt.datatype })|,
        empty
    }
import-schema.element =
    element import-schema {
        extension.atts,
        global.atts,
        attribute namespace { uri.datatype }?,
        attribute _namespace { avt.datatype }?,
        attribute schema-location { uri.datatype }?,
        attribute _schema-location { avt.datatype }?,
        xs_schema.element?
    }
strip-space.element =
    element strip-space {
        extension.atts,
        global.atts,
        (attribute elements { tokens.datatype }
        | attribute _elements { avt.datatype })|,
        empty
    }
preserve-space.element =
    element preserve-space {
        extension.atts,
        global.atts,
        (attribute elements { tokens.datatype }
        | attribute _elements { avt.datatype })|,
        empty
    }
decimal-format.element =
    element decimal-format {
        extension.atts,
        global.atts,
        attribute name { eqname.datatype }?,
        attribute _name { avt.datatype }?,
        attribute decimal-separator { char.datatype }?,
        attribute _decimal-separator { avt.datatype }?,
        attribute grouping-separator { char.datatype }?,
        attribute _grouping-separator { avt.datatype }?,
        attribute infinity { string.datatype }?,
        attribute _infinity { avt.datatype }?,
        attribute minus-sign { char.datatype }?,
        attribute _minus-sign { avt.datatype }?,
        attribute exponent-separator { char.datatype }?,
        attribute _exponent-separator { avt.datatype }?,
        attribute NaN { string.datatype }?,
        attribute _NaN { avt.datatype }?,
        attribute percent { char.datatype }?,
        attribute _percent { avt.datatype }?,
    }

```

```

attribute per-mille { char.datatype }?,
attribute _per-mille { avt.datatype }?,
attribute zero-digit { char.datatype }?,
attribute _zero-digit { avt.datatype }?,
attribute digit { char.datatype }?,
attribute _digit { avt.datatype }?,
attribute pattern-separator { char.datatype }?,
attribute _pattern-separator { avt.datatype }?,
empty
}
template.element =
element template {
  extension.atts,
  global.atts,
  (attribute match { pattern.datatype }
  | attribute _match { avt.datatype }
  | attribute name { eqname.datatype }
  | attribute _name { avt.datatype })|,
  attribute priority { decimal.datatype }?,
  attribute _priority { avt.datatype }?,
  attribute mode { list { '#all' | ('#default' | '#unnamed' | eqname.datatype)* } }?,
  attribute _mode { avt.datatype }?,
  attribute as { sequence-type.datatype }?,
  attribute _as { avt.datatype }?,
  attribute visibility { "public" | "private" | "final" | "abstract" }?,
  attribute _visibility { avt.datatype }?,
  (context-item.element?, param.element*, sequence-constructor.model)
}
apply-templates.element =
element apply-templates {
  extension.atts,
  global.atts,
  attribute select { expression.datatype }?,
  attribute _select { avt.datatype }?,
  attribute mode { (eqname.datatype | '#unnamed' | '#default' | '#current') }?,
  attribute _mode { avt.datatype }?,
  (sort.element | with-param.element)*
}
mode.element =
element mode {
  extension.atts,
  global.atts,
  attribute name { eqname.datatype }?,
  attribute _name { avt.datatype }?,
  attribute streamable { boolean.datatype }?,
  attribute _streamable { avt.datatype }?,
  attribute on-no-match { "deep-copy" | "shallow-copy" | "deep-skip" | "shallow-skip" | "te
attribute _on-no-match { avt.datatype }?,
  attribute on-multiple-match { "use-last" | "fail" }?,
  attribute _on-multiple-match { avt.datatype }?,
  attribute warning-on-no-match { boolean.datatype }?,
  attribute _warning-on-no-match { avt.datatype }?,
  attribute warning-on-multiple-match { boolean.datatype }?,
  attribute _warning-on-multiple-match { avt.datatype }?,
  attribute typed { boolean.datatype | "strict" | "lax" | "unspecified" }?,
  attribute _typed { avt.datatype }?,
  attribute visibility { "public" | "private" | "final" }?,
  attribute _visibility { avt.datatype }?,
  attribute use-accumulators { tokens.datatype }?,
  attribute _use-accumulators { avt.datatype }?,
  empty
}

```

```

    }
context-item.element =
  element context-item {
    extension.atts,
    global.atts,
    attribute as { item-type.datatype }?,
    attribute _as { avt.datatype }?,
    attribute use { "required" | "optional" | "absent" }?,
    attribute _use { avt.datatype }?,
    empty
  }
global-context-item.element =
  element global-context-item {
    extension.atts,
    global.atts,
    attribute as { item-type.datatype }?,
    attribute _as { avt.datatype }?,
    attribute use { "required" | "optional" | "absent" }?,
    attribute _use { avt.datatype }?,
    empty
  }
apply-imports.element =
  element apply-imports {
    extension.atts,
    global.atts,
    with-param.element*
  }
next-match.element =
  element next-match {
    extension.atts,
    global.atts,
    (with-param.element | fallback.element)*
  }
for-each.element =
  element for-each {
    extension.atts,
    global.atts,
    (attribute select { expression.datatype }
     | attribute _select { avt.datatype })|,
    (sort.element*, sequence-constructor.model)
  }
iterate.element =
  element iterate {
    extension.atts,
    global.atts,
    (attribute select { expression.datatype }
     | attribute _select { avt.datatype })|,
    (param.element*, on-completion.element?, sequence-constructor.model)
  }
next-iteration.element =
  element next-iteration {
    extension.atts,
    global.atts,
    (with-param.element*)
  }
break.element =
  element break {
    extension.atts,
    global.atts,
    (attribute select { expression.datatype }
     | attribute _select { avt.datatype })?,
    empty
  }

```

```

sequence-constructor.model
}
on-completion.element =
element on-completion {
    extension.atts,
    global.atts,
    attribute select { expression.datatype }?,
    attribute _select { avt.datatype }?,
    sequence-constructor.model
}
if.element =
element if {
    extension.atts,
    global.atts,
    (attribute test { expression.datatype }
     | attribute _test { avt.datatype })+,
    sequence-constructor.model
}
choose.element =
element choose {
    extension.atts,
    global.atts,
    (when.element+, otherwise.element?)}
when.element =
element when {
    extension.atts,
    global.atts,
    (attribute test { expression.datatype }
     | attribute _test { avt.datatype })+,
    sequence-constructor.model
}
otherwise.element =
element otherwise {
    extension.atts,
    global.atts,
    sequence-constructor.model
}
try.element =
element try {
    extension.atts,
    global.atts,
    attribute select { expression.datatype }?,
    attribute _select { avt.datatype }?,
    attribute rollback-output { boolean.datatype }?,
    attribute _rollback-output { avt.datatype }?,
    (sequence-constructor.model, catch.element, (catch.element | fallback.element)*)
}
catch.element =
element catch {
    extension.atts,
    global.atts,
    attribute errors { tokens.datatype }?,
    attribute _errors { avt.datatype }?,
    attribute select { expression.datatype }?,
    attribute _select { avt.datatype }?,
    sequence-constructor.model
}
variable.element =
element variable {
    extension.atts,

```

```

global.atts,
(attribute name { eqname.datatype }
| attribute _name { avt.datatype })+,
attribute select { expression.datatype }?,
attribute _select { avt.datatype }?,
attribute as { sequence-type.datatype }?,
attribute _as { avt.datatype }?,
attribute static { boolean.datatype }?,
attribute _static { avt.datatype }?,
attribute visibility { "public" | "private" | "final" | "abstract" }?,
attribute _visibility { avt.datatype }?,
sequence-constructor.model
}
param.element =
element param {
extension.atts,
global.atts,
(attribute name { eqname.datatype }
| attribute _name { avt.datatype })+,
attribute select { expression.datatype }?,
attribute _select { avt.datatype }?,
attribute as { sequence-type.datatype }?,
attribute _as { avt.datatype }?,
attribute required { boolean.datatype }?,
attribute _required { avt.datatype }?,
attribute tunnel { boolean.datatype }?,
attribute _tunnel { avt.datatype }?,
attribute static { boolean.datatype }?,
attribute _static { avt.datatype }?,
sequence-constructor.model
}
with-param.element =
element with-param {
extension.atts,
global.atts,
(attribute name { eqname.datatype }
| attribute _name { avt.datatype })+,
attribute select { expression.datatype }?,
attribute _select { avt.datatype }?,
attribute as { sequence-type.datatype }?,
attribute _as { avt.datatype }?,
attribute tunnel { boolean.datatype }?,
attribute _tunnel { avt.datatype }?,
sequence-constructor.model
}
call-template.element =
element call-template {
extension.atts,
global.atts,
(attribute name { eqname.datatype }
| attribute _name { avt.datatype })+,
with-param.element*
}
attribute-set.element =
element attribute-set {
extension.atts,
global.atts,
(attribute name { eqname.datatype }
| attribute _name { avt.datatype })+,
attribute use-attribute-sets { eqnames.datatype }?,
attribute _use-attribute-sets { avt.datatype }?,

```

```

attribute visibility { "public" | "private" | "final" | "abstract" }?,
attribute _visibility { avt.datatype }?,
attribute streamable { boolean.datatype }?,
attribute _streamable { avt.datatype }?,
attribute.element*
}
function.element =
element function {
  extension.atts,
  global.atts,
  (attribute name { eqname.datatype }
  | attribute _name { avt.datatype })|,
  attribute as { sequence-type.datatype }?,
  attribute _as { avt.datatype }?,
  attribute visibility { "public" | "private" | "final" | "abstract" }?,
  attribute _visibility { avt.datatype }?,
  attribute streamability { "unclassified" | "absorbing" | "inspection" | "filter" | "shall"
  attribute _streamability { avt.datatype }?,
  attribute override-extension-function { boolean.datatype }?,
  attribute _override-extension-function { avt.datatype }?,
  attribute override { boolean.datatype }?,
  attribute _override { avt.datatype }?,
  attribute new-each-time { "yes" | "true" | "1" | "no" | "false" | "0" | "maybe" }?,
  attribute _new-each-time { avt.datatype }?,
  attribute cache { boolean.datatype }?,
  attribute _cache { avt.datatype }?,
  (param.element*, sequence-constructor.model)
}
evaluate.element =
element evaluate {
  extension.atts,
  global.atts,
  (attribute xpath { expression.datatype }
  | attribute _xpath { avt.datatype })|,
  attribute as { sequence-type.datatype }?,
  attribute _as { avt.datatype }?,
  attribute base-uri { uri.datatype | avt.datatype }?,
  attribute _base-uri { avt.datatype }?,
  attribute with-params { expression.datatype }?,
  attribute _with-params { avt.datatype }?,
  attribute context-item { expression.datatype }?,
  attribute _context-item { avt.datatype }?,
  attribute namespace-context { expression.datatype }?,
  attribute _namespace-context { avt.datatype }?,
  attribute schema-aware { boolean.datatype | avt.datatype }?,
  attribute _schema-aware { avt.datatype }?,
  (with-param.element | fallback.element)*
}
namespace-alias.element =
element namespace-alias {
  extension.atts,
  global.atts,
  (attribute stylesheet-prefix { prefix.datatype | "#default" }
  | attribute _stylesheet-prefix { avt.datatype })|,
  (attribute result-prefix { prefix.datatype | "#default" }
  | attribute _result-prefix { avt.datatype })|,
  empty
}
element.element =
element element {
  extension.atts,

```

```

global.atts,
(attribute name { qname.datatype | avt.datatype }
| attribute _name { avt.datatype })+,
attribute namespace { uri.datatype | avt.datatype }?,
attribute _namespace { avt.datatype }?,
attribute inherit-namespaces { boolean.datatype }?,
attribute _inherit-namespaces { avt.datatype }?,
attribute use-attribute-sets { eqnames.datatype }?,
attribute _use-attribute-sets { avt.datatype }?,
((attribute type { eqname.datatype }?,
attribute _type { avt.datatype }) |
(attribute validation { "strict" | "lax" | "preserve" | "strip" }?,
attribute _validation { avt.datatype }? )), # type and validation are mutually exclusive
sequence-constructor.model
}
attribute.element =
element attribute {
extension.atts,
global.atts,
(attribute name { qname.datatype | avt.datatype }
| attribute _name { avt.datatype })+,
attribute namespace { uri.datatype | avt.datatype }?,
attribute _namespace { avt.datatype }?,
attribute select { expression.datatype }?,
attribute _select { avt.datatype }?,
attribute separator { string.datatype | avt.datatype }?,
attribute _separator { avt.datatype }?,
((attribute type { eqname.datatype }?,
attribute _type { avt.datatype }) |
(attribute validation { "strict" | "lax" | "preserve" | "strip" }?,
attribute _validation { avt.datatype }? )), # type and validation are mutually exclusive
sequence-constructor.model
}
text.element =
element text {
extension.atts,
global.atts,
attribute disable-output-escaping { boolean.datatype }?,
attribute _disable-output-escaping { avt.datatype }?,
text
}
value-of.element =
element value-of {
extension.atts,
global.atts,
attribute select { expression.datatype }?,
attribute _select { avt.datatype }?,
attribute separator { string.datatype | avt.datatype }?,
attribute _separator { avt.datatype }?,
attribute disable-output-escaping { boolean.datatype }?,
attribute _disable-output-escaping { avt.datatype }?,
sequence-constructor.model
}
document.element =
element document {
extension.atts,
global.atts,
((attribute type { eqname.datatype }?,
attribute _type { avt.datatype }) |
(attribute validation { "strict" | "lax" | "preserve" | "strip" }?,
attribute _validation { avt.datatype }? )), # type and validation are mutually exclusive

```

```

        sequence-constructor.model
    }
processing-instruction.element =
element processing-instruction {
    extension.atts,
    global.atts,
    (attribute name { ncname.datatype | avt.datatype }
    | attribute _name { avt.datatype })+,
    attribute select { expression.datatype }?,
    attribute _select { avt.datatype }?,
    sequence-constructor.model
}
namespace.element =
element namespace {
    extension.atts,
    global.atts,
    (attribute name { ncname.datatype | avt.datatype }
    | attribute _name { avt.datatype })+,
    attribute select { expression.datatype }?,
    attribute _select { avt.datatype }?,
    sequence-constructor.model
}
comment.element =
element comment {
    extension.atts,
    global.atts,
    attribute select { expression.datatype }?,
    attribute _select { avt.datatype }?,
    sequence-constructor.model
}
copy.element =
element copy {
    extension.atts,
    global.atts,
    attribute select { expression.datatype }?,
    attribute _select { avt.datatype }?,
    attribute copy-namespaces { boolean.datatype }?,
    attribute _copy-namespaces { avt.datatype }?,
    attribute inherit-namespaces { boolean.datatype }?,
    attribute _inherit-namespaces { avt.datatype }?,
    attribute use-attribute-sets { eqnames.datatype }?,
    attribute _use-attribute-sets { avt.datatype }?,
    ((attribute type { eqname.datatype }?,
    attribute _type { avt.datatype }?) |
    (attribute validation { "strict" | "lax" | "preserve" | "strip" }?,
    attribute _validation { avt.datatype }?) ), # type and validation are mutually exclusive
    sequence-constructor.model
}
copy-of.element =
element copy-of {
    extension.atts,
    global.atts,
    (attribute select { expression.datatype }
    | attribute _select { avt.datatype })+,
    attribute copy-accumulators { boolean.datatype }?,
    attribute _copy-accumulators { avt.datatype }?,
    attribute copy-namespaces { boolean.datatype }?,
    attribute _copy-namespaces { avt.datatype }?,
    ((attribute type { eqname.datatype }?,
    attribute _type { avt.datatype }?) |
    (attribute validation { "strict" | "lax" | "preserve" | "strip" }?,
    attribute _validation { avt.datatype }?) ),
    sequence-constructor.model
}

```

```

        attribute _validation { avt.datatype }? ), # type and validation are mutually exclusive
        empty
    }
sequence.element =
    element sequence {
        extension.atts,
        global.atts,
        attribute select { expression.datatype }?,
        attribute _select { avt.datatype }?,
        sequence-constructor.model
    }
where-populated.element =
    element where-populated {
        extension.atts,
        global.atts,
        sequence-constructor.model
    }
on-empty.element =
    element on-empty {
        extension.atts,
        global.atts,
        attribute select { expression.datatype }?,
        attribute _select { avt.datatype }?,
        sequence-constructor.model
    }
on-non-empty.element =
    element on-non-empty {
        extension.atts,
        global.atts,
        attribute select { expression.datatype }?,
        attribute _select { avt.datatype }?,
        sequence-constructor.model
    }
number.element =
    element number {
        extension.atts,
        global.atts,
        attribute value { expression.datatype }?,
        attribute _value { avt.datatype }?,
        attribute select { expression.datatype }?,
        attribute _select { avt.datatype }?,
        attribute level { "single" | "multiple" | "any" }?,
        attribute _level { avt.datatype }?,
        attribute count { pattern.datatype }?,
        attribute _count { avt.datatype }?,
        attribute from { pattern.datatype }?,
        attribute _from { avt.datatype }?,
        attribute format { string.datatype | avt.datatype }?,
        attribute _format { avt.datatype }?,
        attribute lang { language.datatype | avt.datatype }?,
        attribute _lang { avt.datatype }?,
        attribute letter-value { "alphabetic" | "traditional" | avt.datatype }?,
        attribute _letter-value { avt.datatype }?,
        attribute ordinal { string.datatype | avt.datatype }?,
        attribute _ordinal { avt.datatype }?,
        attribute start-at { integer.datatype | avt.datatype }?,
        attribute _start-at { avt.datatype }?,
        attribute grouping-separator { char.datatype | avt.datatype }?,
        attribute _grouping-separator { avt.datatype }?,
        attribute grouping-size { integer.datatype | avt.datatype }?,
        attribute _grouping-size { avt.datatype }?,
    }

```

```

    empty
}
sort.element =
element sort {
    extension.atts,
    global.atts,
    attribute select { expression.datatype }?,
    attribute _select { avt.datatype }?,
    attribute lang { language.datatype | avt.datatype }?,
    attribute _lang { avt.datatype }?,
    attribute order { "ascending" | "descending" | avt.datatype }?,
    attribute _order { avt.datatype }?,
    attribute collation { uri.datatype | avt.datatype }?,
    attribute _collation { avt.datatype }?,
    attribute stable { boolean.datatype | avt.datatype }?,
    attribute _stable { avt.datatype }?,
    attribute case-order { "upper-first" | "lower-first" | avt.datatype }?,
    attribute _case-order { avt.datatype }?,
    attribute data-type { "text" | "number" | eqname.datatype | avt.datatype }?,
    attribute _data-type { avt.datatype }?,
    sequence-constructor.model
}
perform-sort.element =
element perform-sort {
    extension.atts,
    global.atts,
    attribute select { expression.datatype }?,
    attribute _select { avt.datatype }?,
    (sort.element+, sequence-constructor.model)
}
for-each-group.element =
element for-each-group {
    extension.atts,
    global.atts,
    (attribute select { expression.datatype }
    | attribute _select { avt.datatype })|,
    ((attribute group-by { expression.datatype }?,
    attribute _group-by { avt.datatype }?) |
    (attribute group-adjacent { expression.datatype }?,
    attribute _group-adjacent { avt.datatype }?) |
    (attribute group-starting-with { pattern.datatype }?,
    attribute _group-starting-with { avt.datatype }?) |
    (attribute group-ending-with { pattern.datatype }?,
    attribute _group-ending-with { avt.datatype }?)),
    attribute composite { boolean.datatype }?,
    attribute _composite { avt.datatype }?,
    attribute collation { uri.datatype | avt.datatype }?,
    attribute _collation { avt.datatype }?,
    (sort.element*, sequence-constructor.model)
}
merge.element =
element merge {
    extension.atts,
    global.atts,
    (merge-source.element+, merge-action.element, fallback.element*)
}
merge-source.element =
element merge-source {
    extension.atts,
    global.atts,
    attribute name { ncname.datatype }?,

```

```

attribute _name { avt.datatype }?,
attribute for-each-item { expression.datatype }?,
attribute _for-each-item { avt.datatype }?,
attribute for-each-stream { expression.datatype }?,
attribute _for-each-stream { avt.datatype }?,
(attribute select { expression.datatype }
| attribute _select { avt.datatype })+,
attribute streamable { boolean.datatype }?,
attribute _streamable { avt.datatype }?,
attribute use-accumulators { tokens.datatype }?,
attribute _use-accumulators { avt.datatype }?,
attribute sort-before-merge { boolean.datatype }?,
attribute _sort-before-merge { avt.datatype }?,
attribute validation { "strict" | "lax" | "preserve" | "strip" }?,
attribute _validation { avt.datatype }?,
attribute type { eqname.datatype }?,
attribute _type { avt.datatype }?,
attribute for-each-source { expression.datatype }?,
attribute _for-each-source { avt.datatype }?,
merge-key.element+
}

merge-key.element =
element merge-key {
  extension.atts,
  global.atts,
  attribute select { expression.datatype }?,
  attribute _select { avt.datatype }?,
  attribute lang { language.datatype | avt.datatype }?,
  attribute _lang { avt.datatype }?,
  attribute order { "ascending" | "descending" | avt.datatype }?,
  attribute _order { avt.datatype }?,
  attribute collation { uri.datatype | avt.datatype }?,
  attribute _collation { avt.datatype }?,
  attribute case-order { "upper-first" | "lower-first" | avt.datatype }?,
  attribute _case-order { avt.datatype }?,
  attribute data-type { "text" | "number" | eqname.datatype | avt.datatype }?,
  attribute _data-type { avt.datatype }?,
  sequence-constructor.model
}
merge-action.element =
element merge-action {
  extension.atts,
  global.atts,
  sequence-constructor.model
}
fork.element =
element fork {
  extension.atts,
  global.atts,
  (fallback.element*, ((sequence.element, fallback.element*)* | (for-each-group.element, fa
})
analyze-string.element =
element analyze-string {
  extension.atts,
  global.atts,
  (attribute select { expression.datatype }
| attribute _select { avt.datatype })+,
  (attribute regex { string.datatype | avt.datatype }
| attribute _regex { avt.datatype })+,
  attribute flags { string.datatype | avt.datatype }?,
  attribute _flags { avt.datatype }?,

```

```

        (matching-substring.element?, non-matching-substring.element?, fallback.element*)
    }
matching-substring.element =
    element matching-substring {
        extension.atts,
        global.atts,
        sequence-constructor.model
    }
non-matching-substring.element =
    element non-matching-substring {
        extension.atts,
        global.atts,
        sequence-constructor.model
    }
source-document.element =
    element source-document {
        extension.atts,
        global.atts,
        (attribute href { uri.datatype | avt.datatype }
        | attribute _href { avt.datatype })?,
        attribute use-accumulators { tokens.datatype }?,
        attribute _use-accumulators { avt.datatype }?,
        ((attribute type { eqname.datatype }?,
        attribute _type { avt.datatype }) |
        (attribute validation { "strict" | "lax" | "preserve" | "strip" }?,
        attribute _validation { avt.datatype }?) ), # type and validation are mutually exclusive
        attribute streamable { boolean.datatype }?,
        attribute _streamable { avt.datatype }?,
        sequence-constructor.model
    }
accumulator.element =
    element accumulator {
        extension.atts,
        global.atts,
        (attribute name { eqname.datatype }
        | attribute _name { avt.datatype }),
        (attribute initial-value { expression.datatype }
        | attribute _initial-value { avt.datatype }),
        attribute as { sequence-type.datatype }?,
        attribute _as { avt.datatype }?,
        attribute streamable { boolean.datatype }?,
        attribute _streamable { avt.datatype }?,
        accumulator-rule.element+
    }
accumulator-rule.element =
    element accumulator-rule {
        extension.atts,
        global.atts,
        (attribute match { pattern.datatype } |
        attribute _match { avt.datatype })?,
        attribute phase { "start" | "end" }?,
        attribute _phase { avt.datatype }?,
        attribute select { expression.datatype }?,
        attribute _select { avt.datatype }?,
        sequence-constructor.model
    }
key.element =
    element key {
        extension.atts,
        global.atts,
        (attribute name { eqname.datatype })

```

```

| attribute _name { avt.datatype }+,
(attribute match { pattern.datatype }
| attribute _match { avt.datatype }+,
attribute use { expression.datatype }?,
attribute _use { avt.datatype }?,
attribute composite { boolean.datatype }?,
attribute _composite { avt.datatype }?,
attribute collation { uri.datatype }?,
attribute _collation { avt.datatype }?,
sequence-constructor.model
}
map.element =
element map {
  extension.atts,
  global.atts,
  sequence-constructor.model
}
map-entry.element =
element map-entry {
  extension.atts,
  global.atts,
  (attribute key { expression.datatype }
| attribute _key { avt.datatype }),
attribute select { expression.datatype }?,
attribute _select { avt.datatype }?,
sequence-constructor.model
}
message.element =
element message {
  extension.atts,
  global.atts,
  attribute select { expression.datatype }?,
  attribute _select { avt.datatype }?,
  attribute terminate { boolean.datatype | avt.datatype }?,
  attribute _terminate { avt.datatype }?,
  attribute error-code { eqname.datatype | avt.datatype }?,
  attribute _error-code { avt.datatype }?,
  sequence-constructor.model
}
assert.element =
element assert {
  extension.atts,
  global.atts,
  (attribute test { expression.datatype }
| attribute _test { avt.datatype })+,
attribute select { expression.datatype }?,
attribute _select { avt.datatype }?,
attribute error-code { eqname.datatype | avt.datatype }?,
attribute _error-code { avt.datatype }?,
sequence-constructor.model
}
fallback.element =
element fallback {
  extension.atts,
  global.atts,
  sequence-constructor.model
}
result-document.element =
element result-document {
  extension.atts,
  global.atts,

```

```

attribute format { eqname.datatype | avt.datatype }?,
attribute _format { avt.datatype }?,
attribute href { uri.datatype | avt.datatype }?,
attribute _href { avt.datatype }?,
((attribute type { eqname.datatype }|,
attribute _type { avt.datatype }) |
(attribute validation { "strict" | "lax" | "preserve" | "strip" }?,
attribute _validation { avt.datatype }?), # type and validation are mutually exclusive
attribute method { "xml" | "html" | "xhtml" | "text" | "json" | "adaptive" | eqname.datat
attribute _method { avt.datatype }?,
attribute allow-duplicate-names { boolean.datatype | avt.datatype }?,
attribute _allow-duplicate-names { avt.datatype }?,
attribute build-tree { boolean.datatype | avt.datatype }?,
attribute _build-tree { avt.datatype }?,
attribute byte-order-mark { boolean.datatype | avt.datatype }?,
attribute _byte-order-mark { avt.datatype }?,
attribute cdata-section-elements { eqnames.datatype | avt.datatype }?,
attribute _cdata-section-elements { avt.datatype }?,
attribute doctype-public { string.datatype | avt.datatype }?,
attribute _doctype-public { avt.datatype }?,
attribute doctype-system { string.datatype | avt.datatype }?,
attribute _doctype-system { avt.datatype }?,
attribute encoding { string.datatype | avt.datatype }?,
attribute _encoding { avt.datatype }?,
attribute escape-uri-attributes { boolean.datatype | avt.datatype }?,
attribute _escape-uri-attributes { avt.datatype }?,
attribute html-version { decimal.datatype | avt.datatype }?,
attribute _html-version { avt.datatype }?,
attribute include-content-type { boolean.datatype | avt.datatype }?,
attribute _include-content-type { avt.datatype }?,
attribute indent { boolean.datatype | avt.datatype }?,
attribute _indent { avt.datatype }?,
attribute item-separator { string.datatype | avt.datatype }?,
attribute _item-separator { avt.datatype }?,
attribute json-node-output-method { "xml" | "html" | "xhtml" | "text" | eqname.datatype | 
attribute _json-node-output-method { avt.datatype }?,
attribute media-type { string.datatype | avt.datatype }?,
attribute _media-type { avt.datatype }?,
attribute normalization-form { "NFC" | "NFD" | "NFKC" | "NFKD" | "fully-normalized" | "no
attribute _normalization-form { avt.datatype }?,
attribute omit-xml-declaration { boolean.datatype | avt.datatype }?,
attribute _omit-xml-declaration { avt.datatype }?,
attribute parameter-document { uri.datatype | avt.datatype }?,
attribute _parameter-document { avt.datatype }?,
attribute standalone { boolean.datatype | "omit" | avt.datatype }?,
attribute _standalone { avt.datatype }?,
attribute suppress-indentation { eqnames.datatype | avt.datatype }?,
attribute _suppress-indentation { avt.datatype }?,
attribute undeclare-prefixes { boolean.datatype | avt.datatype }?,
attribute _undeclare-prefixes { avt.datatype }?,
attribute use-character-maps { eqnames.datatype }?,
attribute _use-character-maps { avt.datatype }?,
attribute output-version { nmtoken.datatype | avt.datatype }?,
attribute _output-version { avt.datatype }?,
sequence-constructor.model
}
output.element =
element output {
extension.atts,
global.atts.except.version,
attribute name { eqname.datatype }?,

```

```

attribute _name { avt.datatype }?,
attribute method { "xml" | "html" | "xhtml" | "text" | "json" | "adaptive" | eqname.datat
attribute _method { avt.datatype }?,
attribute allow-duplicate-names { boolean.datatype }?,
attribute _allow-duplicate-names { avt.datatype }?,
attribute build-tree { boolean.datatype }?,
attribute _build-tree { avt.datatype }?,
attribute byte-order-mark { boolean.datatype }?,
attribute _byte-order-mark { avt.datatype }?,
attribute cdata-section-elements { eqnames.datatype }?,
attribute _cdata-section-elements { avt.datatype }?,
attribute doctype-public { string.datatype }?,
attribute _doctype-public { avt.datatype }?,
attribute doctype-system { string.datatype }?,
attribute _doctype-system { avt.datatype }?,
attribute encoding { string.datatype }?,
attribute _encoding { avt.datatype }?,
attribute escape-uri-attributes { boolean.datatype }?,
attribute _escape-uri-attributes { avt.datatype }?,
attribute html-version { decimal.datatype }?,
attribute _html-version { avt.datatype }?,
attribute include-content-type { boolean.datatype }?,
attribute _include-content-type { avt.datatype }?,
attribute indent { boolean.datatype }?,
attribute _indent { avt.datatype }?,
attribute item-separator { string.datatype }?,
attribute _item-separator { avt.datatype }?,
attribute json-node-output-method { "xml" | "html" | "xhtml" | "text" | eqname.datatype }?
attribute _json-node-output-method { avt.datatype }?,
attribute media-type { string.datatype }?,
attribute _media-type { avt.datatype }?,
attribute normalization-form { "NFC" | "NFD" | "NFKC" | "NFKD" | "fully-normalized" | "no
attribute _normalization-form { avt.datatype }?,
attribute omit-xml-declaration { boolean.datatype }?,
attribute _omit-xml-declaration { avt.datatype }?,
attribute parameter-document { uri.datatype }?,
attribute _parameter-document { avt.datatype }?,
attribute standalone { boolean.datatype | "omit" }?,
attribute _standalone { avt.datatype }?,
attribute suppress-indentation { eqnames.datatype }?,
attribute _suppress-indentation { avt.datatype }?,
attribute undeclare-prefixes { boolean.datatype }?,
attribute _undeclare-prefixes { avt.datatype }?,
attribute use-character-maps { eqnames.datatype }?,
attribute _use-character-maps { avt.datatype }?,
attribute version { nmtoken.datatype }?,
attribute _version { avt.datatype }?,
empty
}
character-map.element =
element character-map {
  extension.atts,
  global.atts,
  (attribute name { eqname.datatype }
  | attribute _name { avt.datatype })+,
  attribute use-character-maps { eqnames.datatype }?,
  attribute _use-character-maps { avt.datatype }?,
  (output-character.element*)
}
output-character.element =
element output-character {

```

```

extension.atts,
global.atts,
(attribute character { char.datatype }
| attribute _character { avt.datatype })+,
(attribute string { string.datatype }
| attribute _string { avt.datatype })+,
empty
}
avt.datatype =
xsd:string
# {
#   pattern =
#     """([^\{\}]\{\}|[^{}])|\{([^\'\\{}]|["^"]*|[^\']*')+}\)*"" # this regexp will not work
#
}


```

I Acknowledgements (Non-Normative)

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

The chair of the XSLT WG is Sharon Adler. The active membership of the XSLT WG during the final stages of the preparation of this specification included:

- Sharon Adler (Chair)
- Anders Berglund
- Carine Bournez (W3C team)
- Abel Braaksma
- Charles Foster
- Florent Georges
- Michael Kay (Editor)
- Jirka Kosek
- Luis Ibhiabor
- Michael Sperberg-McQueen
- Norm Walsh
- Mohamed Zergaoui

The Working Group wishes to acknowledge the contribution of those who have participated in the work at earlier stages, as well as the pioneering work of the developers of STX (see [\[STX\]](#)) which formed an important intellectual input to the design of XSLT 3.0 and demonstrated the feasibility of creating a streaming transformation language based on the core XSLT concept of recursive descent of the source tree using rule-based templates.

The Working Group also wishes to thank external reviewers who have provided feedback during the development of the specification.

J Changes since XSLT 2.0 (Non-Normative)

J.1 Changes in this Specification

1. A stylesheet may now consist of multiple packages. The language specification for packages has been designed with a view to allowing packages to be compiled independently of each other. The specification provides control over the interface offered by a package to other packages; in particular it allows functions, variables, named templates and other components to be declared as public, private, final, or abstract.
2. A new [xsl:mode](#) declaration is added.
 - a. A mode may be declared to be streamable, and rules are given that constrain what the template rules in a streamable mode can do.
 - b. An [xsl:mode](#) declaration may define the action to be taken when there is no matching template rule, and the action to be taken when there are multiple matching template rules.
 - c. An [xsl:mode](#) declaration may indicate that the template rules in a given mode are designed to process typed (schema-validated) nodes only, or untyped nodes only. It may also indicate that element names appearing in match patterns for the mode are only to match elements in the source document that have been validated against the corresponding element declarations in the schema.
 - d. A default mode can be declared for a stylesheet module, making it easier to reuse existing stylesheet modules to construct a composite stylesheet.
3. Several new instructions are introduced with the aim of making it easier to write streamable transformations, although all of these instructions can also be used without streaming:
 - a. The [xsl:source-document](#) instruction is provided to read and process an input document using streaming.
 - b. The [xsl:iterate](#) instruction allows iterative processing of a sequence, with the ability for the processing of one item to depend on the results of processing of previous items, and with the ability to terminate the iteration before all the items in the sequence have been processed.
 - c. The [xsl:merge](#) instruction allows several input sequences to be merged into a single output sequence, based on the value of a merge key.
 - d. The [xsl:fork](#) instruction allows multiple results to be computed during a single pass of a streamed input document.

The [xsl:sequence](#) instruction can now contain a sequence constructor as an alternative to using the `select` attribute. This is primarily for use cases involving [xsl:fork](#).

 - e. New instructions [xsl:where-populated](#), [xsl:on-empty](#), and [xsl:on-non-empty](#) are introduced to allow elements to be generated only when relevant content exists (or does not exist), without requiring the input to be processed more than once.
4. Other changes introduced to facilitate the writing of streamable transformations include:
 - a. The new [top-level](#) declaration [xsl:accumulator](#) is introduced. An accumulator represents information about a node in a document that can be computed during a streamed pass over the document, starting at the start and ending at that node.
 - b. New functions [copy-of](#) and [snapshot](#) are provided, to enable streaming applications to operate in windowing mode, where the input document is divided into a sequence of small subtrees processed one at a time.
5. Some further new instructions are provided, unrelated to streaming:
 - a. The [xsl:try](#) instruction allows recovery from dynamic errors.
 - b. A new [xsl:evaluate](#) instruction is provided, to allow evaluation of XPath expressions constructed dynamically from strings, or read from a source document.

- c. The [xsl:assert](#) instruction allows arbitrary assertions about the state of variables or the input document, improving testability and robustness.
- 6. Static global variables and parameters can be declared. These act as compile-time constants. The values of static variables can be used in initializing other static variables, or in `[xsl:]use-when` attributes, or in *shadow attributes*. Shadow attributes allow any attribute of an XSLT instruction or declaration to be parameterized by reference to static variables and parameters.
- 7. Text nodes within a sequence constructor may now contain [text value templates](#) (XPath expressions enclosed in curly brackets), if this is enabled by setting `expand-text="yes">` on an enclosing element. This reduces the verbosity of code written to generate boilerplate text with variable inserts.
- 8. The syntax of [patterns](#) has been generalized. Patterns may now match any item (not only nodes). In consequence, [xsl:apply-templates](#) can now process sequences of atomic values as well as nodes, and [xsl:for-each-group](#) with the `group-starting-with` and `group-ending-with` options can also process atomic sequences. As a further consequence, the items in the [initial match selection](#) supplied when initiating a transformation are no longer required to be nodes.
- 9. A new datatype, called a **map**, has been introduced, together with supporting functions, operators, and type syntax. Maps allow more complex data structures to be created than is possible using atomic values and nodes alone. This has particular applications to streamed processing: since a streamed application can visit each node of its primary input document only once, it often needs more advanced data structures to retain what it has already seen in the document.
- 10. Miscellaneous changes to existing instructions and declarations include:
 - a. The regular expression supplied to the [xsl:analyze-string](#) instruction is now permitted to be one that matches a zero-length string.
 - b. The [xsl:copy](#) instruction now has a `select` attribute, which is convenient when it is used inside a function where there is no context item.
 - c. Composite keys are supported in [xsl:for-each-group](#).
 - d. Two new attributes have been added to [xsl:function](#) to provide increased scope for optimization: `new-each-time` and `cache`. The first indicates whether the identity of nodes created by the function is significant to the application; the second indicates whether the function is to cache its results (memoization).
 - e. The `override` attribute of [xsl:function](#) is renamed `override-extension-function`, retaining the old name as a deprecated synonym.
 - f. The rule requiring [xsl:import](#) declarations to precede all other declarations in a stylesheet module has been removed.
 - g. Composite keys are supported in [xsl:key](#).
 - h. A new attribute on [xsl:message](#) allows specification of an error code.
 - i. The rules for handling conflicts between [xsl:strip-space](#) and [xsl:preserve-space](#) have changed. A conflict that can be detected statically is now signaled as a static error; a run-time conflict between two declarations having the same precedence and priority is now resolved by taking whichever comes last in declaration order.
 - j. An [xsl:template](#) declaration may contain an [xsl:context-item](#) element to declare the required type of the context item when the template is called.
 - k. An empty [xsl:value-of](#) instruction with no `select` attribute is now permitted; its effect is to construct a zero-length text node.

- l. The [xsl:variable](#) and [xsl:param](#) elements may now specify `static="yes"`, denoting that the variable is available statically (informally, “at compile time”). Static variables and parameters make the [xsl:]use-when mechanism more useful, especially in conjunction with [xsl:assert](#).
11. New functions are available to import and export data in JSON format.
12. A [basic XSLT Processor](#) now recognizes all the built-in types defined in XML Schema.
13. A [basic XSLT Processor](#) will now accept the attribute `validation="lax"` and interpret it in the same way as a schema-aware processor when there is no schema component available to perform the validation.
14. Some functions, including [generate-id](#)^{FO30}, [format-date](#)^{FO30}, [format-datetime](#)^{FO30}, [format-number](#)^{FO30}, [format-time](#)^{FO30}, and [unparsed-text](#)^{FO30} have been moved from this specification to the Functions and Operators specification, to make them available in other host languages.
15. The rule that effectively prevented references to external documents in [xsl:]use-when expressions has been removed.
16. A default value is defined for the named template to be used when initiating a transformation (specifically, [xsl:initial-template](#)).
17. Serialization to HTML5 and XHTML5 is supported. To this end, a new serialization parameter `html-version` is provided in [xsl:output](#) and [xsl:result-document](#). Other new serialization parameters include: `item-separator`, `json-node-output-method`, `parameter-document`, `suppress-indentation`.
18. The concept of recoverable dynamic errors has been dropped. Of the remaining recoverable dynamic errors, some are no longer errors, and others are now situations where the behavior of the processor is [implementation-dependent](#). The adjective *non-recoverable* in describing other dynamic errors becomes redundant and has therefore been dropped (the term was in any case misleading since the introduction of a try/catch mechanism). Error codes of the form XTREnnnn have been renumbered XTDEnnnn. Dynamic errors occurring during pattern evaluation are always masked (they cause the pattern to report a non-match.)
19. A family of collation URIs is defined for selecting collations based on the Unicode Collation Algorithm.
20. The effect of specifying the type `xs:untyped` or `xs:untypedAtomic` when validating by type is now defined.
21. The set of constructs that set [temporary output state](#) has been reduced, and no longer includes instructions that create nodes, such as [xsl:attribute](#) and [xsl:value-of](#). However, [xsl:merge-key](#) has been added to the list.
22. The possibilities for invocation of a stylesheet have been expanded; they now include the ability to directly execute a stylesheet function; to supply parameters to the initial template; and to return the results of the invoked template or function as a raw value, without construction of a result tree.

J.2 Changes in Other Related Specifications

A number of changes affecting XSLT 3.0 have been made in other related specifications. Some of the more significant changes are as follows:

1. A number of new functions have been defined whose aim is to facilitate streaming. These include [unparsed-text-lines](#)^{FO30}, [innermost](#)^{FO30}, [outermost](#)^{FO30}.

2. XPath 3.0 supports a subset of the `let` expression from XQuery.
3. XPath 3.0 supports function items as first-class values (functions can, for example, be bound to variables and passed as parameters to other functions.)
4. XPath 3.0 supports a new syntax for writing expanded names using the namespace URI and local part only, avoiding the need to create a static context that binds namespace prefixes. This is intended to be particularly useful when XPath expressions are software-generated. Complementing this, a new function `path`^{FO30} is available to generate a (namespace-context-independent) path to any node that can subsequently be evaluated using the `xsl:evaluate` instruction, or otherwise.

K Changes since the Candidate Recommendation of 19 November 2015 (Non-Normative)

This section contains a list of changes that were made between the first Candidate Recommendation in November 2015 and the second Candidate Recommendation in February 2017. Design changes affecting the syntax or semantics of the XSLT language are marked (**). Minor changes to edge cases, and cases where rules have been supplied that were previously missing, are marked (*). Other changes may be considered editorial: these include corrections to examples, addition of non-normative notes, removal of ambiguities and inconsistencies, and textual clarifications. Changes that are purely typographical are not listed.

1. [Bug29234](#): If explicit packages are used, then the initial mode used when a stylesheet is invoked (like an initial template or initial function) must now be declared as public. (**)
2. [Bug29256](#): Clarified that it is an error for the top-level package to contain abstract components, whether or not the components are referenced. (*)
3. [Bug29340](#): Streamability rules for XPath expressions now include the XPath 3.1 production number as well as the XPath 3.0 production number.
4. [Bug29342](#): The streamability rules for XPath 3.1 arrow expressions did not cover dynamic function calls.
5. [Bug29351](#): An error code has been allocated for the type error that occurs when `xsl:evaluate/@with-params` is not a map, or is a map of the wrong type. (*)
6. [Bug29392](#): Defined how support for the `serialize`^{FO30} function relates to the optional [serialization feature](#). (*)
7. [Bug29425](#): The syntax summary now marks the `xsl:result-document` attributes `method` and `json-node-output-method` as attribute value templates, bringing it into line with the prose.
8. [Bug29431](#): The rules for returning the principal and secondary results of a transformation, and in particular the interaction of `build-tree` and `item-separator`, have been clarified. (*)
9. [Bug29436](#): The list of instructions in 5.7 that return the results of a contained sequence constructor without alteration has been corrected and made non-normative.
10. [Bug29441](#): The term [extension function](#) has been more carefully defined.
11. [Bug29442](#): Part of the text on evaluating [sequence constructors](#) has been rewritten to improve clarity.
12. [Bug29445](#): The summary of the rules for selecting a separator in [5.7.2 Constructing Simple Content](#) has been made more complete.
13. [Bug29449](#): The section on streamability of dynamic function calls now provides non-normative advice on the use of this construct in conjunction with maps and arrays.

14. [Bug29453](#): An `xsl:use-package` declaration may appear in an included stylesheet module but not in an imported stylesheet module. (**)
15. [Bug29455](#): Added to the list of items that are considered empty (now `vacuous`) by the `xsl:on-empty` and `xsl:on-non-empty` instructions, for example to include zero-length strings, and arrays consisting entirely of vacuous items. (**)
16. [Bug29459](#): Clarified the rules for streamability of arrow expressions. (*)
17. [Bug29460](#): The introduction to the concept of packages now mentions that an implementation might allow packages to be written in other languages (for example, XQuery).
18. [Bug29461](#): Clarified how the concept of “static base URI” should be interpreted in situations where the source stylesheet is not available at execution time. (*)
19. [Bug29468](#): Modified the rules for the default visibility of overriding components. (*)
20. [Bug29473](#): Removed a misleading suggestion that the default visibility of overriding components is always `private`; this is not the case for `xsl:param`.
21. [Bug29474](#): There was an incorrect suggestion that `xsl:original` could be used to refer to a declaration overridden using the traditional mechanism of import precedence. This has been removed.
22. [Bug29478](#): In response to usability feedback, `xsl:expose` and `xsl:accept` now allow the value `component="*"` to mean “all kinds of component”. (**)
23. [Bug29480](#): Defined that the focus for evaluating attribute sets referenced by `xsl:copy` is the same as the focus for evaluating the contained sequence constructor (*)
24. [Bug29482](#): The Working Group decided not to change the streamability rules to make a particular use case involving `xsl:copy guaranteed-streamable`, but instead to add a note explaining how to rewrite this use case in a streamable way.
25. [Bug29492](#): Simplified the rules for streamability of attribute sets. Attribute sets can no longer be `consuming`. (*)
26. [Bug29502](#): The streamability rules for `xsl:fork` were incomplete for the case where the instruction has an `xsl:for-each-group` child. (*)
27. [Bug29507](#): Clarified that a striding expression such as `(/a/b, $doc/a/b)` can deliver a mix of streamed and unstreamed nodes and that the result is not necessarily in document order.
28. [Bug29544](#): Clarified that whitespace stripping does not apply to the trees returned by functions such as `parse-xml`^{FO30} and `parse-xml-fragment`^{FO30}. (*)
29. [Bug29558](#): When the namespace used for the XML representation of JSON was changed to `http://www.w3.org/2005/xpath-functions`, one reference to the old namespace was not updated.
30. [Bug29574](#): By default a public component in a used package now becomes private in the using package. This also affects the treatment of abstract components; as part of this change, the keyword `visibility="absent"` is dropped. (**)
31. [Bug29588](#): In the `xml-to-json` function, map keys are now compared after normalizing escape sequences to determine whether duplicates exist. (*)
32. [Bug29602](#): Changes made to the `xml-to-json` in the XPath 3.1 project have been retrofitted to this document. The changes include the detailed rules for escaping special characters, and the adoption of uniform conventions for type-checking and conversion of parameter options. (**)

33. [Bug29604](#): Corrected a throwaway remark in the text of an example concerning how to compute multiple aggregate values in a single pass of the input.
34. [Bug29660](#): The function `map:remove` can now remove multiple entries from a map in a single call. (**)
35. [Bug29665](#): The `xml-to-json` now escapes a solidus (/ becomes \/), which is useful when the resulting JSON is embedded in HTML. (**)
36. [Bug29666](#): Added a note to clarify how the concept of stylesheet levels relates to packages.
37. [Bug29667](#): Added a note to confirm that it is not intrinsically an error to have two `xsl:use-package` declarations that reference the same package.
38. [Bug29669](#): Introductory material describing the XSLT processing model has been rewritten for clarity.
39. [Bug29675](#): The rules for determining the context item static type in a global variable declaration take account of the declared type in the `xsl:global-context-item` declaration if available. (*)
40. [Bug29686](#): Clarified the rules for compatibility of types when overriding functions. (*)
41. [Bug29690](#): The requirement that a streaming processor should always use streaming if requested is relaxed if for example (a) the input is supplied as a tree in memory, or (b) the processor is able to determine that the input document is too small for streaming to give any benefit.
42. [Bug29692](#): Clarified how stripping of type annotations and whitespace text nodes works when the rules vary by package. (*)
43. [Bug29696](#): The `global context item` is not streamable. (**)
44. [Bug29697](#): Clarified where calls on `current-merge-group` and `current-merge-key` can be used when `xsl:merge` instructions are nested. (*)
45. [Bug29698](#): Corrected several mentions of the `streamable` attribute of `xsl:merge`; the attribute actually appears on `xsl:merge-source`.
46. [Bug29699](#): Clarified that for the purpose of error XTSE3085, only `xsl:template` elements with a `match` attribute are relevant. (*)
47. [Bug29709](#): Some `xsl:merge` examples used obsolete syntax from an earlier working draft.
48. [Bug29710](#): Used more precise terminology in some of the rules defining the streamability of stylesheet functions.
49. [Bug29712](#): Streamable stylesheet functions declared with streamability absorbing, shallow-descent, or deep-descent now allow the function body to be motionless (previously it had to be consuming). (*)
50. [Bug29716](#): Defined additional situations where the `current template rule` is cleared. (*)
51. [Bug29723](#): The `map:merge` function takes a second argument to control how duplicate keys are handled. (**)
52. [Bug29732](#): The rules for streamable stylesheet functions have been refined. In most cases the argument must now be a single node rather than a sequence of nodes, and constraints on the variable reference have been rewritten as constraints on the function signature. (**)
53. [Bug29733](#): An example for an absorbing stylesheet function was correct, but the explanation for why it was correct was misleading.
54. [Bug29738](#): The vague term “streamable stylesheet” is no longer used.
55. [Bug29743](#): A new function `map:find` is provided to allow recursive searching of nested maps. (**)
56. [Bug29747](#): The `xsl:stream` instruction has been generalized to handle both streamed and unstreamed processing, and it has accordingly been renamed `xsl:source-document`, and has a `streamable` attribute.

(**)

57. [Bug29752](#): Improved an example where accumulators are used to compute a word count, to give a more realistic real-world result.
58. [Bug29763](#): Added rules concerning the effect of `xsl:expose` and `xsl:accept` on `xsl:param` declarations (which are always public). (*)
59. [Bug29768](#): The operand usages for the `map:for-each` and `map:merge` functions have been corrected (affecting the streamability of these functions in the unusual case where the functions are called with references to streamed nodes). (*)
60. [Bug29790](#): The sample stylesheet for the `xml-to-json` function has been changed to avoid using a reserved namespace.
61. [Bug29793](#): Added a note confirming that the `input-type-annotations` attribute has no effect on an `xsl:source-document` instruction when the `type` or `validation` attributes are present.
62. [Bug29796](#): A note has been added pointing out that keys only allow searching within a tree that is rooted at a document node.
63. [Bug29802](#): Clarified the text describing the function of the `global context item`.
64. [Bug29803](#): Having been dropped from `xsl:global-context-item`, the ability to control which accumulators are used on the initial match selection has been moved to `xsl:mode`. (**)
65. [Bug29804](#): The `for-each-stream` attribute of `xsl:merge-source` has been generalized to handle both streamed and unstreamed processing, and it has accordingly been renamed `for-each-source`; streaming of the merge input is controlled using the `streamable` attribute. (**)
66. [Bug29805](#): The `use-accumulators` attribute of `xsl:source-document` now applies whether or not the instruction is declared streamable. (**)
67. [Bug29811](#): Clarified what error code should be used for a particular error involving static variables. (*)
68. [Bug29813](#): A section has been added explaining how to handle dynamic errors that occur during the evaluation of accumulators. (*)
69. [Bug29814](#): Clarified that XPath comments can appear (only) in attributes of type expression, pattern, item-type, or sequence-type. (*)
70. [Bug29819](#): Dropped the use of the term “core functions” in favour of more precise wording.
71. [Bug29827](#): Clarified the rules defining which modes are eligible to be used as the `initial mode` when a stylesheet is first invoked. (*)
72. [Bug29853](#): The result of the `collation-key` function is now always `xs:base64Binary`, making the comparison semantics unambiguous and context-independent. (**)
73. [Bug29865](#): A new parameter `maxVariable` is added to UCA collation URIs, to define which groups of characters (such as whitespace and punctuation) are ignored, or treated as less significant, when comparing strings. In addition, interoperable defaults are defined for most of the collation parameters. (**)
74. [Bug29860](#), [Bug29861](#), [Bug29862](#), [Bug29865](#): Fixed errors in the schema for XSLT 3.0 stylesheets resulting from changes logged elsewhere, notable the renaming of `xsl:stream` to `xsl:source-document`.
75. [Bug29866](#): Changed the definition of type `EQName` in the schema for XSLT 3.0 stylesheets to be more restrictive.
76. [Bug29880](#): Implementations may impose limits on the values used in a package version number, and minimum values for those limits have been defined. (*)

77. [Bug29887](#): Changed an assertion against `xsl:for-each-group` in the schema for XSLT 3.0 stylesheets to be allow for the possibility of shadow attributes. (*)
78. [Bug29889](#): Added an introductory section concerning streaming of non-XML data.
79. [Bug29917](#): The `xml-to-json` function now allows the top-level element of the input to have a `key` attribute (which is ignored), so that it can successfully process any subtree of the output of `json-to-xml`. (*)
80. [Bug29919](#): A `use-when` attribute on `xsl:package` works the same way as on `xsl:stylesheet` and `xsl:transform`. (*)
81. [Bug29920](#): The rules for shadow attributes have been rewritten to avoid using the undefined term *target attribute*.
82. [Bug29927](#): Clarified that facilities for disabling `xsl:evaluate` are implementation-defined.
83. [Bug29933](#): In line with other serialization parameters, the detail of what `undeclare-prefixes` does is now delegated to the serialization specification.
84. [Bug29960](#): Processors are now allowed to provide a mode of operation in which there is no requirement to report static errors in code that is never executed. (*)
85. [Bug29975](#): Added a non-normative summary of the rules affecting validation of `xml:id` attributes.
86. [Bug29978](#): Rules relating the the permitted children of `xsl:stylesheet` apply also to `xsl:package`. (*)
87. [Bug29980](#): Editorial improvements to the definition of error XTSE0760. (*)
88. [Bug29981](#): Relaxed the rule requiring the tunnel parameters on an overriding template to be identical to those on the overridden template. (**)
89. [Bug29982](#): Expanded the note explaining the rationale and use cases for tunnel parameters.
90. [Bug29983](#): The justification and explanation for the streamability of *scanning expression* such as `//section/head` has been rewritten for clarity; and the term itself is now defined in terms of the rules for *motionless patterns* since the two concepts are very closely aligned. (*)
91. [Bug29988](#): Clarified that in the `xs:QName` values returned by `available-system-properties`, the prefix part of the QName is implementation-dependent.
92. [Bug30002](#): Rectified the omission of `xsl:function/@cache` in the schema for XSLT 3.0.
93. [Bug30032](#): Refined the static typing rules for axis steps to take account of the axis as well as the node test. (*)
94. [Bug30033](#): Refined the rules for streamability of the `current` function. (*)
95. [Bug30034](#): Corrected a note concerning the streamability of `xsl:choose` to match the normative rules.
96. [Bug30036](#): `document-node(E)` sequence type tests cause streaming difficulties for `treat as` expressions just as they do for `instance of` expressions. (*)
97. [Bug30049](#): Supplied missing rules regarding the dynamic context for evaluation of XPath expressions (both static expressions and expressions evaluated using `xsl:evaluate`), especially as regards the named functions available in the dynamic context for `function-lookup`^{FO30}. (*)
98. [Bug30056](#): Corrected the expected output of an example of streamed grouping.

A non-normative Relax NG schema for XSLT 3.0 has been added to [H Schemas for XSLT 3.0 Stylesheets](#).

L Changes since the Candidate Recommendation of 7 February 2017 (Non-Normative)

This section contains a list of changes that have been made. Trivial typographic errors and changes to non-normative front and back matter are not listed.

1. [Bug30060](#): An example purporting to show streamable use of `xsl:iterate` was not in fact guaranteed-streamable, and has been corrected by injecting a call of the `copy-of` function.
2. [Bug30064](#): Added a Note to explain why certain path expressions (such as `./section/head`) are not guaranteed-streamable.

M Changes since the Proposed Recommendation of 18 April 2017 (Non-Normative)

This section contains a list of changes that have been made. Trivial typographic errors and changes to non-normative front and back matter are not listed.

1. [Bug30089](#): A non-normative note in 2.3.6.1 wrongly stated that serializing an array would raise an error. The serialization specification is clear that this is not the case.
2. [Bug30090](#): A non-normative note in 3.2 referred to the "four standard serialization methods". With the 3.1 version of the serialization specification, there are now six standard methods.
3. [Bug30091](#): A non-normative note in 3.5.4 advised users to solve a particular problem by using the instruction `<xsl:apply-templates/>`. Better advice would be to use `<xsl:apply-templates select=". "/>`
4. [Bug30093](#): An example in [5.5.1 Examples of Patterns](#) (carried over unchanged from the XSLT 2.0 specification) gave incorrect semantics for the pattern `//para`.
5. [Bug30094](#): In [9.1 Variables](#) the term `local variable` is used. The definition of this term appears in [9.8 Local Variables and Parameters](#). But rather than linking to the definition, the former section gave an incomplete explanation of its meaning.
6. [Bug30095](#): In [9.9 Scope of Variables](#), the scope of global variables was described without making clear that the discussion was in the context of a single package; a note has been added explaining that the rules for cross-package visibility are defined elsewhere.
7. [Bug30099](#): In the proforma for the `start-at` attribute of `xsl:number` (appearing in both section 12 and Appendix D), the type of the attribute was given as `integer` although the normative description of the syntax and semantics of the attribute makes clear that a whitespace-separated sequence of integers is permitted. (For technical reasons this change is not color-highlighted.)
8. [Bug30100](#): A non-normative note in section 27.4 referred to version "3.0", in a context that made no sense unless this is read as version "2.0".
9. [Bug30109](#): A non-normative note has been added to explain that as a consequence of the general rules for constructing simple content, the `disable-output-escaping` attribute has no effect when writing attributes, comments or processing instructions.

N Incompatibilities with XSLT 2.0 (Non-Normative)

This section lists all known incompatibilities with XSLT 2.0, that is, situations where a stylesheet that is error-free according to the XSLT 2.0 specification and where all elements have an effective version of 2.0 or less, will produce different results depending on whether it is run under an XSLT 2.0 processor or an XSLT 3.0 processor.

1. XSLT 2.0 gave implementations freedom what to do when a node selected by [`xsl:apply-templates`](#) matched more than one [`template rule`](#). XSLT 3.0 is more prescriptive in this situation. The behavior prescribed in XSLT 3.0 (selecting the template rule that is last in [`declaration order`](#)) is compatible with the action of some XSLT 2.0 processors but not necessarily others.
 2. It is now a static error if the same `NameTest` appears in both an [`xsl:strip-space`](#) and an [`xsl:preserve-space`](#) declaration with the same precedence and priority. Previously this was a dynamic error, and processors were allowed to recover from the error.
 3. The current group and current grouping key are now absent rather than empty when not in use, which means that attempting to refer to them in this state gives a dynamic error.
 4. As a consequence of functions such as [`format-date`^{FO30}](#) moving from this specification to [\[Functions and Operators 3.0\]](#), error codes associated with these functions have changed.
 5. The concept of recoverable dynamic errors has been dropped. Of the remaining recoverable dynamic errors, some are no longer errors, and others are now situations where the behavior of the processor is [`implementation-dependent`](#). Error codes of the form XTREnnnn have been renumbered XTDEnnnn.
 6. In previous versions of the specification, the [`element-available`](#) function when applied to names in the XSLT namespace was defined to return `false` in the case of XSLT elements other than instructions. (Actual practice in implementations was not always consistent with this rule). In XSLT 3.0 the rules have been changed so that it returns `true` for the names of such elements, bringing the specification of the function into line with the intuitive meaning of its name.
 7. (This is not strictly speaking an incompatibility, as conforming XSLT 2.0 stylesheets will continue to function correctly without error. It can be considered as migration advice, a warning that care is needed when introducing new XSLT 3.0 features.)
- When a function or template has a parameter with a declared type of `item()`, it should not assume (as it could in XSLT 2.0) that when the supplied item is not a node, it must be an atomic value, and vice versa. In XSLT 3.0 there is a third option: it might be a function. Functions and templates that fail to cater for this possibility may fail with a type error if the caller supplies a function as the relevant parameter value.
8. XSLT 1.0 and 2.0 required the `grouping-size` attribute of [`xsl:number`](#) to be a **number** (a term which in other contexts was defined to mean any decimal value), but no interpretation was provided for non-integer values. XSLT 3.0 requires the value to be an integer.
 9. In XPath 3.0, the rules for matching node tests of the form `element(*, U)` or `attribute(*, U)` have changed in the case where U is a union type. Specifically, an element or attribute whose type annotation is a member type of U will now match such a node test, whereas in XPath 2.0 it did not. Since the semantics of XSLT pattern matching are based on the XPath rules for matching node tests, this change affects which template rules are chosen to match a node when the match patterns use one of these forms.
 10. The handling of XSLT version numbers that do not correspond to any published specification has changed. An example is `version="1.1"` (which is sometimes encountered because it was used in examples in a popular book). XSLT 2.0 requires processors to treat all values less than 2.0 as if 1.0 were specified. XSLT 3.0 recommends that processors reject such a value as a static error.
 11. XSLT 3.0 disallows the use of certain [`reserved namespaces`](#) in extension functions and extension instructions, and in the `[xsl:]extension-element-prefixes` attribute.

Note:

For example, it becomes an error to write `extension-element-prefixes="xs"` where the prefix `xs` is bound to the XML Schema namespace. Such an attribute is occasionally seen where `exclude-result-prefixes` was probably intended.

