XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)

W3C Recommendation 14 December 2010 (revised 13 April 2015)

This version:
http://www.w3.org/TR/2010/REC-xpath-functions-20101214/
Latest version:
http://www.w3.org/TR/xpath-functions/
Previous versions:
http://www.w3.org/TR/2009/PER-xpath-functions-20090421/,
http://www.w3.org/TR/2007/REC-xpath-functions-20070123/
Editors:
Ashok Malhotra, Oracle Corporation <ashok.malhotra@alum.mit.edu>
Jim Melton, Oracle Corporation <jim.melton@oracle.com>
Norman Walsh, Mark Logic <Norman.Walsh@marklogic.com>
Michael Kay, Saxonica <http://www.saxonica.com/> - Second Edition

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

See also **translations**.

This document is also available in these non-normative formats: XML and Change markings relative to first edition.

Note: *This paragraph is informative.* This document is currently not maintained. This document remains available on the W3C's Technical Report web page for reference and use by interested parties. Readers are advised that no further maintenance (including correction of reported errors) is planned for this document. Readers interested in the most recent version of the XQuery and XPath Functions and Operators specification are encouraged to refer to http://www.w3.org/TR/xpath-functions-3/.

Abstract

This document defines constructor functions, operators and functions on the datatypes defined in [XML Schema Part 2: Datatypes Second Edition] and the datatypes defined in [XQuery 1.0 and XPath 2.0 Data Model]. It also discusses functions and operators on nodes and node sequences as defined in the [XQuery 1.0 and XPath 2.0 Data Model]. These functions and operators are defined for use in [XML Path Language (XPath) 2.0], [XQuery 1.0: An XML Query Language] and

[XSL Transformations (XSLT) Version 2.0] and other related XML standards. The signatures and summaries of functions defined in this document are available at: http://www.w3.org/2005/xpath-functions/.

Status of this Document

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

This is one document in a set of eight documents that are being progressed to Edited Recommendation together (XPath 2.0, XQuery 1.0, XQueryX 1.0, XSLT 2.0, Data Model (XDM), Functions and Operators, Formal Semantics, Serialization).

This document, published on 14 December 2010, is an Edited Recommendation of the W3C. This second edition is not a new version of this specification; its purpose is to clarify a number of issues that have become apparent since the first edition was published. All of these clarifications (excepting trivial editorial fixes) have been published in a separate errata document, and published in a Proposed Edited Recommendation in April, 2009. The changes are summarized in an appendix.

This document has been jointly developed by the W3C XML Query Working Group and the W3C XSL Working Group, each of which is part of the XML Activity.

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.

This document incorporates changes made against the Recommendation of 23 January 2007 that resolve all errata known at the date of publication. A list of the errata that have been applied, with links to the Bugzilla database, is provided in **F Changes since the First Edition**. The version of this document with change highlighting indicates where the textual changes have been made, and cross-references each textual change to the erratum where it originated. This document supersedes the first edition.

This specification is designed to be referred to normatively from other specifications defining a host language for it; it is not intended to be implemented outside a host language. The implementability of this specification has been tested in the context of its normative inclusion in host languages defined by the XQuery 1.0 and XSLT 2.0 specifications; see the XQuery 1.0 implementation report and the XSLT 2.0 implementation report (member-only) for details.

Table of Contents

Appendices

1 Introduction

The purpose of this document is to catalog the functions and operators required for XPath 2.0, XML Query 1.0 and XSLT 2.0. The exact syntax used to invoke these functions and operators is specified in [XML Path Language (XPath) 2.0], [XQuery 1.0: An XML Query Language] and [XSL Transformations (XSLT) Version 2.0].

This document defines constructor functions and functions that take typed values as arguments. Some of the functions define the semantics of operators discussed in [XQuery 1.0: An XML Query Language].

[XML Schema Part 2: Datatypes Second Edition] defines a number of primitive and derived datatypes, collectively known as built-in datatypes. This document defines functions and operations on these datatypes as well as the datatypes defined in Section 2.6 Types*DM* of the [XQuery 1.0 and XPath 2.0 Data Model]. These functions and operations are defined for use in [XML Path Language (XPath) 2.0], [XQuery 1.0: An XML Query Language] and [XSL Transformations (XSLT) Version 2.0] and related XML standards. This document also discusses functions and operators on nodes and node sequences as defined in the [XQuery 1.0 and XPath 2.0 Data Model] for use in [XML Path Language (XPath) 2.0], [XQuery 1.0: An XML Query Language] and [XSL Transformations (XSLT) Version 2.0] and other related XML standards.

References to specific sections of some of the above documents are indicated by cross-document links in this document. Each such link consists of a pointer to a specific section followed a superscript specifying the linked document. The superscripts have the following meanings: 'XQ' [XQuery 1.0: An XML Query Language], 'XT' [XSL Transformations (XSLT) Version 2.0], 'XP' [XML Path Language (XPath) 2.0], 'DM' [XQuery 1.0 and XPath 2.0 Data Model] and 'FS' [XQuery 1.0 and XPath 2.0 Formal Semantics].

1.1 Conformance

The Functions and Operators specification is intended primarily as a component that can be used by other specifications. Therefore, Functions and Operators relies on specifications that use it (such as [XML Path Language (XPath) 2.0], [XSL Transformations (XSLT) Version 2.0] and [XQuery 1.0: An XML Query Language]) to specify conformance criteria for their respective environments.

Authors of conformance criteria for the use of the Functions and Operators should pay particular attention to the following features:

- It is ·implementation-defined· which version of Unicode is supported, but it is recommended that the most recent version of Unicode be used.
- Support for XML 1.0 and XML 1.1 by the datatypes used in Functions and Operators.

  Note:

  At the time of writing there is no published version of XML Schema that references the XML 1.1 specifications. This means that datatypes such as `xs:NCName` and `xs:ID` are constrained by the XML 1.0 rules. Authors of conformance requirements for the use of Functions and Operators should state clearly the implications for conformance of any changes to the rules in later versions of XML Schema.

In this document, text labeled as an example or as a Note is provided for explanatory purposes and is not normative.

1.2 Namespaces and Prefixes

The functions and operators discussed in this document are contained in one of three namespaces (see [Namespaces in XML]) and referenced using an `xs:QName`. The datatypes and constructor functions for the built-in datatypes defined in [XML Schema Part 2: Datatypes Second Edition] and in Section 2.6 Types^DM of [XQuery 1.0 and XPath 2.0 Data Model] and discussed in **5 Constructor Functions** are in the XML Schema namespace, `http://www.w3.org/2001/XMLSchema`, and named in this document using the `xs` prefix. The namespace prefix used in this document for functions that are available to users is `fn`. Operator functions are named with the prefix `op`.

This document uses the prefix `err` to represent the namespace URI `http://www.w3.org/2005/xqt-errors`, which is the namespace for all XPath and XQuery error codes and messages. This namespace prefix is not predeclared and its use in this document is not normative.

The namespace prefix used for the functions, datatypes and errors can vary, as long as the prefix is bound to the correct URI.

The URIs of the namespaces and the default prefixes associated with them are:

- `http://www.w3.org/2001/XMLSchema` for constructors -- associated with `xs`.
- `http://www.w3.org/2005/xpath-functions` for functions -- associated with `fn`.
- `http://www.w3.org/2005/xqt-errors` -- associated with `err`.

Note:

The namespace URI associated with the `err` prefix is not expected to change from one version of this document to another. The contents of this namespace may be extended to allow additional errors to be returned.

The functions defined with an `fn` prefix are callable by the user. Functions defined with the `op` prefix are described here to underpin the definitions of the operators in [XML Path Language (XPath) 2.0], [XQuery 1.0: An XML Query Language] and [XSL Transformations (XSLT) Version 2.0]. These functions are not available directly to users, and there is no requirement that implementations should actually provide these functions. For this reason, no namespace is associated with the `op` prefix. For example, multiplication is generally associated with the `*` operator, but it is described as a function in this document:

```
op:numeric-multiply($arg1 as numeric, $arg2 as numeric) as numeric
```

1.3 Function Overloading

In general, the specifications named above do not support function overloading in the sense that functions that have multiple signatures with the same name and the same number of parameters are not supported. Consequently, there are no such overloaded functions in this document except for legacy [XML Path Language (XPath) Version 1.0] functions such as `fn:string()`, which accepts a single parameter of a variety of types. In addition, it should be noted that the functions defined in **6 Functions and Operators on Numerics** that accept `numeric` parameters accept arguments of type `xs:integer`, `xs:decimal`, `xs:float` or `xs:double`. See **1.4 Function Signatures and Descriptions**. Operators such as "+" may be overloaded. This document does define some functions with more than one signature with the same name and different number of parameters. User-defined functions with more than one signature with the same name and different number of parameters are also supported.

1.4 Function Signatures and Descriptions

Each function is defined by specifying its signature, a description of the return type and each of the parameters and its semantics. For many functions, examples are included to illustrate their use.

Each function's signature is presented in a form like this:

```
fn:function-name($parameter-name as parameter-type, ...) as return-type
```

In this notation, **function-name**, in bold-face, is the name of the function whose signature is being specified. If the function takes no parameters, then the name is followed by an empty parameter list: "`()`"; otherwise, the name is followed by a parenthesized list of parameter declarations, each declaration specifies the static type of the parameter, in italics, and a descriptive, but non-normative, name. If there are two or more parameter declarations, they are separated by a comma. The *`return-type`* , also in italics, specifies the static type of the value returned by the function. The dynamic type returned by the function is the same as its static type or derived from the static type. All parameter types and return types are specified using the SequenceType notation defined in Section 2.5.3 SequenceType Syntax^XP.

In some cases the word " `numeric` " is used in function signatures as a shorthand to indicate the four numeric types: `xs:integer`, `xs:decimal`, `xs:float` and `xs:double`. For example, a function with the signature:

```
fn:numeric-function($arg as numeric) as ...
```

represents the following four function signatures:

```
fn:numeric-function($arg as xs:integer) as ...
```

```
fn:numeric-function($arg as xs:decimal) as ...
```

```
fn:numeric-function($arg as xs:float) as ...
```

```
fn:numeric-function($arg as xs:double) as ...
```

For most functions there is an initial paragraph describing what the function does followed by semantic rules. These rules are meant to be followed in the order that they appear in this document.

In some cases, the static type returned by a function depends on the type(s) of its argument(s). These special functions are indicated by using ***bold italics*** for the return type. The semantic rules specifying the type of the value returned are documented in the function definition. The rules are described more formally in Section 7.2 Standard functions with specific static typing rules^FS.

The function name is a `QName` as defined in [XML Schema Part 2: Datatypes Second Edition] and must adhere to its syntactic conventions. Following [XML Path Language (XPath) Version 1.0], function names are composed of English words separated by hyphens,"-". If a function name contains a [XML Schema Part 2: Datatypes Second Edition] datatype name, it may have intercapitalized spelling and is used in the function name as such. For example, `fn:timezone-from-dateTime`.

Rules for passing parameters to operators are described in the relevant sections of [XQuery 1.0: An XML Query Language] and [XML Path Language (XPath) 2.0]. For example, the rules for passing parameters to arithmetic operators are described in Section 3.4 Arithmetic Expressions^XP. Specifically, rules for parameters of type `xs:untypedAtomic` and the empty sequence are specified in this section.

As is customary, the parameter type name indicates that the function or operator accepts arguments of that type, or types derived from it, in that position. This is called *subtype substitution* (See Section 2.5.4 SequenceType Matching^XP). In addition, numeric type instances and instances of type `xs:anyURI` can be promoted to produce an argument of the required type. (See Section B.1 Type Promotion^XP).

1. *Subtype Substitution*: A derived type may substitute for its base type. In particular, `xs:integer` may be used where `xs:decimal` is expected.
2. *Numeric Type Promotion*: `xs:decimal` may be promoted to `xs:float` or `xs:double`. Promotion to `xs:double` should be done directly, not via `xs:float`, to avoid loss of precision.
3. *anyURI Type Promotion*: A value of type `xs:anyURI` can be promoted to the type `xs:string`.

Some functions accept a single value or the empty sequence as an argument and some may return a single value or the empty sequence. This is indicated in the function signature by following the parameter or return type name with a question mark: "`?`", indicating that either a single value or the empty sequence must appear. See below.

```
fn:function-name($parameter-name as parameter-type?) as return-type?
```

Note that this function signature is different from a signature in which the parameter is omitted. See, for example, the two signatures for `fn:string()`. In the first signature, the parameter is omitted and the argument defaults to the context item, referred to as ".". In the second signature, the argument must be present but may be the empty sequence, referred to as "()."

Some functions accept a sequence of zero or more values as an argument. This is indicated by following the name of type of the items in the sequence with `*`. The sequence may contain zero or more items of the named type. For example, the function below accepts a sequence of `xs:double` and returns a `xs:double` or the empty sequence.

```
fn:median($arg as xs:double*) as xs:double?
```

1.5 Namespace Terminology

This document uses the phrase "namespace URI" to identify the concept identified in [Namespaces in XML] as "namespace name", and the phrase "local name" to identify the concept identified in [Namespaces in XML] as "local part".

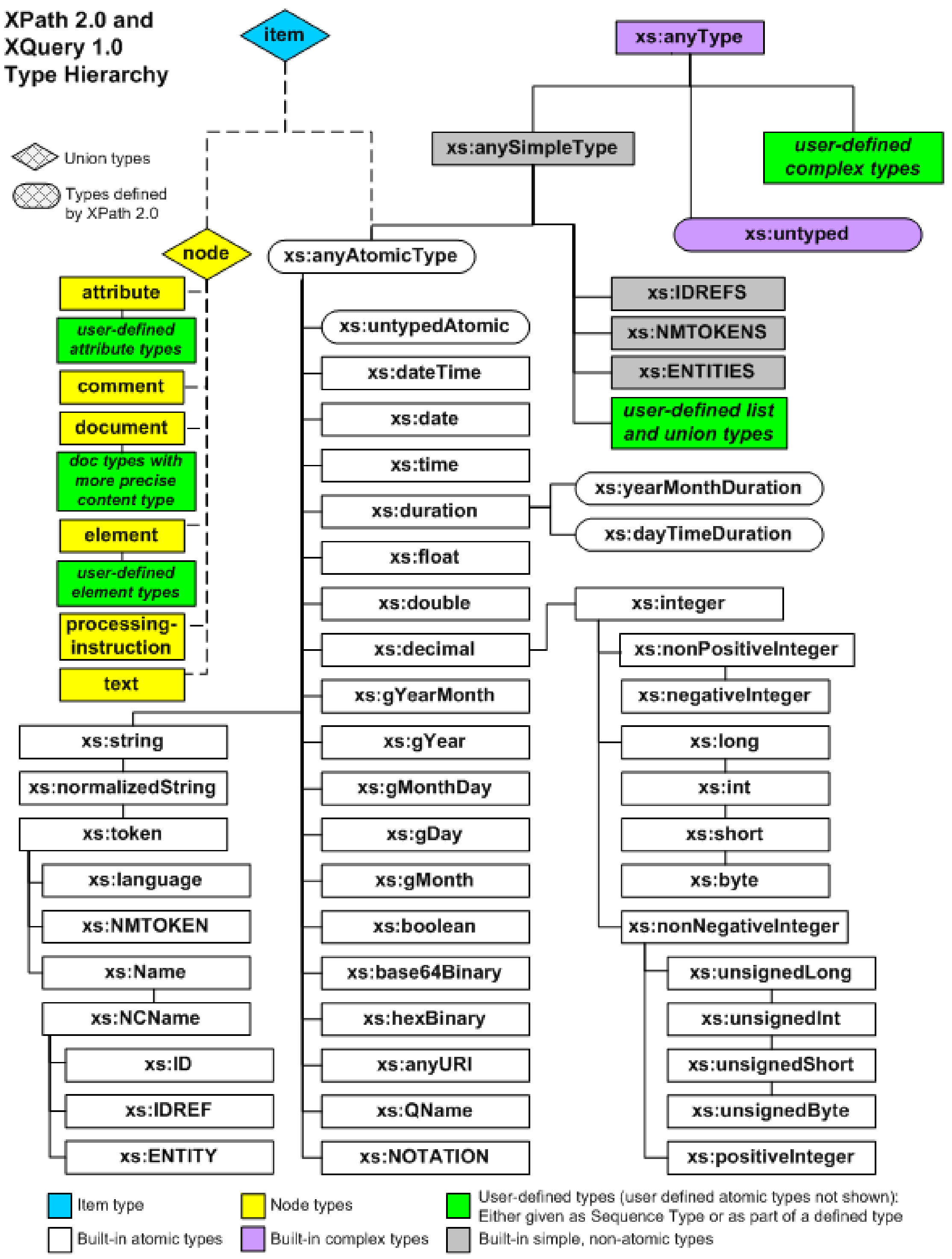
It also uses the term "expanded-QName" defined below.

[Definition] Expanded-QName

> An expanded-QName is a pair of values consisting of a namespace URI and a local name. They belong to the value space of the [XML Schema Part 2: Datatypes Second Edition] datatype `xs:QName`. When this document refers to `xs:QName` we always mean the value space, i.e. a namespace URI, local name pair (and not the lexical space referring to constructs of the form prefix:local-name).

1.6 Type Hierarchy

The diagram below shows the types for which functions are defined in this document. These include the built-in types defined by [XML Schema Part 2: Datatypes Second Edition] (shown on the right) as well as types defined in [XQuery 1.0 and XPath 2.0 Data Model] (shown on the left). Solid lines connect a base datatype above to a derived datatype.`xs:IDREFS`, `xs:NMTOKENS`, `xs:ENTITIES` and `user-defined list and union types` are special types in that these types are lists or unions rather than true subtypes. Dashed lines connect a union type above with its component types below.



The information in the above diagram is reproduced below in tabular form. For ease of presentation the information is divided into three tables. The first table shows the top three layers of the hierarchy starting at `xs:anyType`. The second table shows the types derived from `xs:anyAtomicType`. The third table shows the types defined in [XQuery 1.0 and XPath 2.0 Data Model]

Each type whose name is indented is derived from the type whose name appears nearest above it with one less level of indentation.

xs:anyType	
	user-defined complex types

	xs:untyped	
	xs:anySimpleType	
		user-defined list and union types
		xs:IDREFS
		xs:NMTOKENS
		xs:ENTITIES
		xs:anyAtomicType

The table below shows the datatypes derived from `xs:anyAtomicType`. This includes all the [XML Schema Part 2: Datatypes Second Edition] built-in datatypes as well as the two totally ordered subtypes of duration defined in Section 2.6 Types^DM.

Each type whose name is indented is derived from the type whose name appears nearest above it with one less level of indentation.

xs:untypedAtomic						
xs:dateTime						
xs:date						
xs:time						
xs:duration						
	xs:yearMonthDuration					
	xs:dayTimeDuration					
xs:float						
xs:double						
xs:decimal						
	xs:integer					
		xs:nonPositiveInteger				
			xs:negativeInteger			
		xs:long				
			xs:int			
				xs:short		
					xs:byte	
		xs:nonNegativeInteger				
			xs:unsignedLong			
				xs:unsignedInt		
					xs:unsignedShort	
						xs:unsignedByte
			xs:positiveInteger			
xs:gYearMonth						
xs:gYear						
xs:gMonthDay						
xs:gDay						
xs:gMonth						
xs:string						
	xs:normalizedString					
		xs:token				
			xs:language			
			xs:NMTOKEN			
			xs:Name			
				xs:NCName		
					xs:ID	
					xs:IDREF	
					xs:ENTITY	
xs:boolean						
xs:base64Binary						
xs:hexBinary						
xs:anyURI						
xs:QName						
xs:NOTATION						

The table below shows the type hierarchy for the types introduced in [XQuery 1.0 and XPath 2.0 Data Model]. For these types, each type whose name is indented is a component of the union type whose name appears nearest above with one less level of indentation.

item	
	xs:anyAtomicType

	node		
		attribute	
			user-defined attribute types
		comment	
		document	
			user-defined document types
		element	
			user-defined element types
		processing-instruction	
		text	

1.7 Terminology

The terminology used to describe the functions and operators on [XML Schema Part 2: Datatypes Second Edition] is defined in the body of this specification. The terms defined in the following list are used in building those definitions:

[Definition] for compatibility

A feature of this specification included to ensure that implementations that use this feature remain compatible with [XML Path Language (XPath) Version 1.0]

[Definition] may

Conforming documents and processors are permitted to, but need not, behave as described.

[Definition] must

Conforming documents and processors are required to behave as described; otherwise, they are either non-conformant or else in error.

[Definition] implementation-defined

Possibly differing between implementations, but specified and documented by the implementor for each particular implementation.

[Definition] implementation-dependent

Possibly differing between implementations, but not specified by this or other W3C specification, and not required to be specified by the implementor for any particular implementation.

[Definition] execution scope

The scope over which any two calls on a function would be executed. In XSLT, it applies to any two calls on the function executed during the same transformation. In XQuery, it applies to any two calls executed during the evaluation of a top-level expression i.e. an expression not contained in any other expression. In other contexts, the scope is specified by the host environment that invokes the function library.

[Definition] stable

Most of the functions in the core library have the property that calling the same function twice within an ·execution scope· with the same arguments returns the same result: these functions are said to be **stable**. This category includes a number of functions such as `fn:doc()`, `fn:collection()`, `fn:current-dateTime()`, `fn:current-date` and `fn:current-time()` whose result depends on the external environment. Where the function returns nodes, stability means that the returned nodes are identical, not merely equal and are returned in the same order.

Note:

in the case of `fn:collection()` and `fn:doc()`, the requirement for stability may be relaxed: see the function definitions for details.

Some other functions, for example `fn:position()` and `fn:last()`, depend on the dynamic context and may, therefore, produce different results each time they are called. These functions are said to be **contextual**.

[Definition] URI and URI reference

Within this specification, the term "URI" refers to Universal Resource Identifiers as defined in [RFC 3986] and extended in [RFC 3987] with a new name "IRI". 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: Datatypes Second Edition]. Note that this means, in practice, that where this specification requires a "URI Reference", an IRI as defined in [RFC 3987] will be accepted, provided that other relevant specifications also permit an IRI. The term URI has been retained in preference to IRI to avoid introducing new names for concepts such as "Base URI" that are defined or referenced across the whole family of XML specifications. Note also that the definition of `xs:anyURI` is a wider definition than the definition in [RFC 3987]; for example it does not require non-ASCII characters to be escaped.

2 Accessors

Accessors and their semantics are described in [XQuery 1.0 and XPath 2.0 Data Model]. Some of these accessors are exposed to the user through the functions described below.

Function	Accessor	Accepts	Returns
`fn:node-name`	`node-name`	an optional node	zero or one `xs:QName`
`fn:nilled`	`nilled`	a node	an optional `xs:boolean`
`fn:string`	`string-value`	an optional item or no argument	`xs:string`
`fn:data`	`typed-value`	zero or more items	a sequence of atomic values
`fn:base-uri`	`base-uri`	an optional node or no argument	zero or one `xs:anyURI`
`fn:document-uri`	`document-uri`	an optional node	zero or one `xs:anyURI`

2.1 fn:node-name

```
fn:node-name($arg as node()?) as xs:QName?
```

Summary: Returns an expanded-QName for node kinds that can have names. For other kinds of nodes it returns the empty sequence. If `$arg` is the empty sequence, the empty sequence is returned.

2.2 fn:nilled

```
fn:nilled($arg as node()?) as xs:boolean?
```

Summary: Returns an `xs:boolean` indicating whether the argument node is "nilled". If the argument is not an element node, returns the empty sequence. If the argument is the empty sequence, returns the empty sequence.

2.3 fn:string

```
fn:string() as xs:string
fn:string($arg as item()?) as xs:string
```

Summary: Returns the value of `$arg` represented as a `xs:string`. If no argument is supplied, the context item (`.`) is used as the default argument. The behavior of the function if the argument is omitted is exactly the same as if the context item had been passed as the argument.

If the context item is undefined, error [err:XPDY0002]XP is raised.

If `$arg` is the empty sequence, the zero-length string is returned.

If `$arg` is a node, the function returns the string-value of the node, as obtained using the `dm:string-value` accessor defined in the Section 5.13 string-value AccessorDM.

If `$arg` is an atomic value, then the function returns the same string as is returned by the expression " `$arg` cast as `xs:string` " (see **17 Casting**).

2.4 fn:data

```
fn:data($arg as item()*) as xs:anyAtomicType*
```

Summary: `fn:data` takes a sequence of items and returns a sequence of atomic values.

The result of `fn:data` is the sequence of atomic values produced by applying the following rules to each item in `$arg`:

- If the item is an atomic value, it is returned.
- If the item is a node:
  - If the node does not have a typed value an error is raised [err:FOTY0012].
  - Otherwise, `fn:data()` returns the typed value of the node as defined by the accessor function `dm:typed-value` in Section 5.15 typed-value AccessorDM.

2.5 fn:base-uri

```
fn:base-uri() as xs:anyURI?
```

```
fn:base-uri($arg as node()?) as xs:anyURI?
```

Summary: Returns the value of the base-uri URI property for `$arg` as defined by the accessor function `dm:base-uri()` for that kind of node in Section 5.2 base-uri AccessorDM. If `$arg` is not specified, the behavior is identical to calling the function with the context item (`.`) as argument. The following errors may be raised: if the context item is undefined [err:XPDY0002]XP; if the context item is not a node [err:XPTY0004]XP.

If `$arg` is the empty sequence, the empty sequence is returned.

Document, element and processing-instruction nodes have a base-uri property which may be empty. The base-uri property of all other node types is the empty sequence. The value of the base-uri property is returned if it exists and is not empty. Otherwise, if the node has a parent, the value of `dm:base-uri()` applied to its parent is returned, recursively. If the node does not have a parent, or if the recursive ascent up the ancestor chain encounters a node whose base-uri property is empty and it does not have a parent, the empty sequence is returned.

See also `fn:static-base-uri`.

2.6 fn:document-uri

```
fn:document-uri($arg as node()?) as xs:anyURI?
```

Summary: Returns the value of the document-uri property for `$arg` as defined by the `dm:document-uri` accessor function defined in Section 6.1.2 Accessors^DM.

If `$arg` is the empty sequence, the empty sequence is returned.

Returns the empty sequence if the node is not a document node. Otherwise, returns the value of the `dm:document-uri` accessor of the document node.

In the case of a document node `$D` returned by the `fn:doc` function, or a document node at the root of a tree containing a node returned by the `fn:collection` function, it will always be true that either `fn:document-uri($D)` returns the empty sequence, or that the following expression is true: `fn:doc(fn:document-uri($D))` is `$D`. It is implementation-defined whether this guarantee also holds for document nodes obtained by other means, for example a document node passed as the initial context node of a query or transformation.

3 The Error Function

In this document, as well as in [XQuery 1.0: An XML Query Language], [XML Path Language (XPath) 2.0], and [XQuery 1.0 and XPath 2.0 Formal Semantics], the phrase "an error is raised" is used. Raising an error is equivalent to invoking the `fn:error` function defined in this section with the provided error code.

The above phrase is normally accompanied by specification of a specific error, to wit: "an error is raised [*error code*]". Each error defined in this document is identified by an `xs:QName` that is in the `http://www.w3.org/2005/xqt-errors` namespace, represented in this document by the `err` prefix. It is this `xs:QName` that is actually passed as an argument to the `fn:error` function invocation. Invocation of this function raises an error. For a more detailed treatment of error handing, see Section 2.3.3 Handling Dynamic Errors^XP and Section 7.2.9 The fn:error function^FS.

The `fn:error` function is a general function that may be invoked as above but may also be invoked from [XQuery 1.0: An XML Query Language] or [XML Path Language (XPath) 2.0] applications with, for example, an `xs:QName` argument.

```
fn:error() as none
```

```
fn:error($error as xs:QName) as none
```

```
fn:error($error as xs:QName?, $description as xs:string) as none
```

```
fn:error($error        as xs:QName?,
         $description  as xs:string,
         $error-object as item()*) as none
```

Summary: The `fn:error` function raises an error. While this function never returns a value, an error is returned to the external processing environment as an `xs:anyURI` or an `xs:QName`. The error `xs:anyURI` is derived from the error `xs:QName`. An error `xs:QName` with namespace URI NS and local part LP will be returned as the `xs:anyURI` NS#LP. The method by which the `xs:anyURI` or `xs:QName` is returned to the external processing environment is ·implementation dependent·.

If an invocation provides `$description` and `$error-object`, then these values may also be returned to the external processing environment. The method by which these values are provided to the external environment is ·implementation dependent·.

> **Note:**
>
> The value of the `$description` parameter may need to be localized.

Note that "none" is a special type defined in [XQuery 1.0 and XPath 2.0 Formal Semantics] and is not available to the user. It indicates that the function never returns and ensures that it has the correct static type.

If `fn:error` is invoked with no arguments, then its behavior is the same as the invocation of the following expression:

```
fn:error(fn:QName('http://www.w3.org/2005/xqt-errors', 'err:FOER0000'))
```

If the first argument in the third or fourth signature is the empty sequence it is assumed to be the `xs:QName` constructed by:

```
fn:QName('http://www.w3.org/2005/xqt-errors', 'err:FOER0000')
```

3.1 Examples

- `fn:error()` returns `http://www.w3.org/2005/xqt-errors#FOER0000` (or the corresponding `xs:QName`) to the external processing environment.
- `fn:error(fn:QName('http://www.example.com/HR', 'myerr:toohighsal'), 'Does not apply because salary is too high')` returns `http://www.example.com/HR#toohighsal` and the `xs:string` `"Does not apply because salary is too high"` (or the corresponding `xs:QName`) to the external processing environment.

4 The Trace Function

```
fn:trace($value as item()*, $label as xs:string) as item()*
```

Summary: Provides an execution trace intended to be used in debugging queries.

The input `$value` is returned, unchanged, as the result of the function. In addition, the inputs `$value`, converted to an `xs:string`, and `$label` may be directed to a trace data set. The destination of the trace output is ·implementation-defined·. The format of the trace output is ·implementation dependent·. The ordering of output from invocations of the `fn:trace()` function is ·implementation dependent·.

4.1 Examples

- Consider a situation in which a user wants to investigate the actual value passed to a function. Assume that in a particular execution, `$v` is an `xs:decimal` with value `124.84`. Writing `fn:trace($v, 'the value of $v is:')` will put the strings `"124.84"` and `"the value of $v is:"` in the trace data set in implementation dependent order.

5 Constructor Functions

5.1 Constructor Functions for XML Schema Built-in Types

Every built-in atomic type that is defined in [XML Schema Part 2: Datatypes Second Edition], except `xs:anyAtomicType` and `xs:NOTATION`, has an associated constructor function. `xs:untypedAtomic`, defined in Section 2.6 Types^DM and the two derived types `xs:yearMonthDuration` and `xs:dayTimeDuration` defined in Section 2.6 Types^DM also have associated constructor functions.

A constructor function is not defined for `xs:anyAtomicType` as there are no atomic values with type annotation `xs:anyAtomicType` at runtime, although this can be a statically inferred type. A constructor function is not defined for `xs:NOTATION` since it is defined as an abstract type in [XML Schema Part 2: Datatypes Second Edition]. If the static context (See Section 2.1.1 Static Context^XP) contains a type derived from `xs:NOTATION` then a constructor function is defined for it. See **5.4 Constructor Functions for User-Defined Types**.

The form of the constructor function for a type *prefix:TYPE* is:

```
prefix:TYPE($arg as xs:anyAtomicType?) as prefix:TYPE?
```

If `$arg` is the empty sequence, the empty sequence is returned. For example, the signature of the constructor function corresponding to the `xs:unsignedInt` type defined in [XML Schema Part 2: Datatypes Second Edition] is:

```
xs:unsignedInt($arg as xs:anyAtomicType?) as xs:unsignedInt?
```

Invoking the constructor function `xs:unsignedInt(12)` returns the `xs:unsignedInt` value 12. Another invocation of that constructor function that returns the same `xs:unsignedInt` value is `xs:unsignedInt("12")`. The same result would also be returned if the constructor function were to be invoked with a node that had a typed value equal to the `xs:unsignedInt` 12. The standard features described in Section 2.4.2 Atomization^XP would 'atomize' the node to extract its typed value and then call the constructor with that value. If the value passed to a constructor is illegal for the datatype to be constructed, an error is raised [err:FORG0001].

The semantics of the constructor function " `xs:TYPE(arg)` " are identical to the semantics of " `arg cast as xs:TYPE?` ". See **17 Casting**.

If the argument to a constructor function is a literal, the result of the function may be evaluated statically; if an error is found during such evaluation, it may be reported as a static error.

Special rules apply to constructor functions for `xs:QName` and types derived from `xs:QName` and `xs:NOTATION`. See **5.3 Constructor Functions for xs:QName and xs:NOTATION**.

The following constructor functions for the built-in types are supported:

- `xs:string($arg as xs:anyAtomicType?) as xs:string?`
- `xs:boolean($arg as xs:anyAtomicType?) as xs:boolean?`
- `xs:decimal($arg as xs:anyAtomicType?) as xs:decimal?`
- `xs:float($arg as xs:anyAtomicType?) as xs:float?`

  Implementations ·may· return negative zero for `xs:float("-0.0E0")`. [XML Schema Part 2: Datatypes Second Edition] does not distinguish between the values positive zero and negative zero.

- **xs:double**($arg as *xs:anyAtomicType?*) as *xs:double?*

  Implementations ·may· return negative zero for `xs:double("-0.0E0")`. [XML Schema Part 2: Datatypes Second Edition] does not distinguish between the values positive zero and negative zero.

- **xs:duration**($arg as *xs:anyAtomicType?*) as *xs:duration?*
- **xs:dateTime**($arg as *xs:anyAtomicType?*) as *xs:dateTime?*
- **xs:time**($arg as *xs:anyAtomicType?*) as *xs:time?*
- **xs:date**($arg as *xs:anyAtomicType?*) as *xs:date?*
- **xs:gYearMonth**($arg as *xs:anyAtomicType?*) as *xs:gYearMonth?*
- **xs:gYear**($arg as *xs:anyAtomicType?*) as *xs:gYear?*
- **xs:gMonthDay**($arg as *xs:anyAtomicType?*) as *xs:gMonthDay?*
- **xs:gDay**($arg as *xs:anyAtomicType?*) as *xs:gDay?*
- **xs:gMonth**($arg as *xs:anyAtomicType?*) as *xs:gMonth?*
- **xs:hexBinary**($arg as *xs:anyAtomicType?*) as *xs:hexBinary?*
- **xs:base64Binary**($arg as *xs:anyAtomicType?*) as *xs:base64Binary?*
- **xs:anyURI**($arg as *xs:anyAtomicType?*) as *xs:anyURI?*
- **xs:QName**($arg as *xs:anyAtomicType*) as *xs:QName?*

  See **5.3 Constructor Functions for xs:QName and xs:NOTATION** for special rules.

- **xs:normalizedString**($arg as *xs:anyAtomicType?*) as *xs:normalizedString?*
- **xs:token**($arg as *xs:anyAtomicType?*) as *xs:token?*
- **xs:language**($arg as *xs:anyAtomicType?*) as *xs:language?*
- **xs:NMTOKEN**($arg as *xs:anyAtomicType?*) as *xs:NMTOKEN?*
- **xs:Name**($arg as *xs:anyAtomicType?*) as *xs:Name?*
- **xs:NCName**($arg as *xs:anyAtomicType?*) as *xs:NCName?*
- **xs:ID**($arg as *xs:anyAtomicType?*) as *xs:ID?*
- **xs:IDREF**($arg as *xs:anyAtomicType?*) as *xs:IDREF?*
- **xs:ENTITY**($arg as *xs:anyAtomicType?*) as *xs:ENTITY?*

  See **17.4.1 Casting to xs:ENTITY** for rules related to constructing values of type `xs:ENTITY` and types derived from it.

- **xs:integer**($arg as *xs:anyAtomicType?*) as *xs:integer?*
- **xs:nonPositiveInteger**($arg as *xs:anyAtomicType?*) as *xs:nonPositiveInteger?*
- **xs:negativeInteger**($arg as *xs:anyAtomicType?*) as *xs:negativeInteger?*
- **xs:long**($arg as *xs:anyAtomicType?*) as *xs:long?*
- **xs:int**($arg as *xs:anyAtomicType?*) as *xs:int?*
- **xs:short**($arg as *xs:anyAtomicType?*) as *xs:short?*
- **xs:byte**($arg as *xs:anyAtomicType?*) as *xs:byte?*
- **xs:nonNegativeInteger**($arg as *xs:anyAtomicType?*) as *xs:nonNegativeInteger?*
- **xs:unsignedLong**($arg as *xs:anyAtomicType?*) as *xs:unsignedLong?*
- **xs:unsignedInt**($arg as *xs:anyAtomicType?*) as *xs:unsignedInt?*
- **xs:unsignedShort**($arg as *xs:anyAtomicType?*) as *xs:unsignedShort?*
- **xs:unsignedByte**($arg as *xs:anyAtomicType?*) as *xs:unsignedByte?*
- **xs:positiveInteger**($arg as *xs:anyAtomicType?*) as *xs:positiveInteger?*

- **xs:yearMonthDuration**($arg as *xs:anyAtomicType?*) as *xs:yearMonthDuration?*
- **xs:dayTimeDuration**($arg as *xs:anyAtomicType?*) as *xs:dayTimeDuration?*
- **xs:untypedAtomic**($arg as *xs:anyAtomicType?*) as *xs:untypedAtomic?*

5.2 A Special Constructor Function for xs:dateTime

A special constructor function is provided for constructing a `xs:dateTime` value from a `xs:date` value and a `xs:time` value.

fn:dateTime($arg1 as *xs:date?*, $arg2 as *xs:time?*) as *xs:dateTime?*

The result `xs:dateTime` has a date component whose value is equal to `$arg1` and a time component whose value is equal to `$arg2`. The result is the empty sequence if either of the parameters is the empty sequence.

The timezone of the result is computed as follows:

- If neither argument has a timezone, the result has no timezone.
- If exactly one of the arguments has a timezone, or if both arguments have the same timezone, the result has this timezone.
- If the two arguments have different timezones, an error is raised:[err:FORG0008]

5.2.1 Examples

- `fn:dateTime(xs:date("1999-12-31"), xs:time("12:00:00"))` returns `xs:dateTime("1999-12-31T12:00:00")`.
- `fn:dateTime(xs:date("1999-12-31"), xs:time("24:00:00"))` returns `xs:dateTime("1999-12-31T00:00:00")` because `"24:00:00"` is an alternate lexical form for `"00:00:00"`.

5.3 Constructor Functions for xs:QName and xs:NOTATION

Special rules apply to constructor functions for the types `xs:QName` and `xs:NOTATION`, for two reasons:

- The lexical representation of these types uses namespace prefixes, whose meaning is context-dependent.
- Values cannot belong directly to the type `xs:NOTATION`, only to its subtypes.

These constraints result in the following restrictions:

- Conversion from an `xs:string` to a value of type `xs:QName`, a type derived from `xs:QName` or a type derived from `xs:NOTATION` is permitted only if the `xs:string` is written as a string literal. This applies whether the conversion is expressed using a constructor function or using the "cast as" syntax. Such a conversion can be regarded as a pseudo-function, which is always evaluated statically. It is also permitted for these constructors and casts to take a dynamically-supplied argument in the normal manner, but as the casting table (see **17.1 Casting from primitive types to primitive types**) indicates, the only arguments that are supported in this case are values of type `xs:QName` or `xs:NOTATION` respectively.
- There is no constructor function for `xs:NOTATION`. Constructors are defined, however, for `xs:QName`, for types derived from `xs:QName`, and for types derived from `xs:NOTATION`.

When converting from an `xs:string`, the prefix within the lexical `xs:QName` supplied as the argument is resolved to a namespace URI using the statically known namespaces from the static context. If the lexical `xs:QName` has no prefix, the namespace URI of the resulting expanded-QName is the default element/type namespace from the static context. Components of the static context are discussed in Section 2.1.1 Static Context*XP*. A static error is raised [err:FONS0004] if the prefix is not bound in the static context. As described in Section 2.1 Terminology*DM*, the supplied prefix is retained as part of the expanded-QName value.

5.4 Constructor Functions for User-Defined Types

For every atomic type in the static context (See Section 2.1.1 Static Context*XP*) that is derived from a primitive type, there is a constructor function (whose name is the same as the name of the type) whose effect is to create a value of that type from the supplied argument. The rules for constructing user-defined types are defined in the same way as the rules for constructing built-in derived types discussed in **5.1 Constructor Functions for XML Schema Built-in Types**.

Special rules apply to constructor functions for types derived from `xs:QName` and `xs:NOTATION`. See **5.3 Constructor Functions for xs:QName and xs:NOTATION**.

Consider a situation where the static context contains a type called `hatSize` defined in a schema whose target namespace is bound to the prefix `my`. In such a case the constructor function:

```
my:hatSize($arg as xs:anyAtomicType?) as my:hatSize?
```

is available to users.

To construct an instance of an atomic type that is not in a namespace, it is necessary to use a cast expression or undeclare the default function namespace. For example, if the user-defined type `apple` is derived from `xs:integer` but is not in a namespace, an instance of this type can be constructed as follows using a cast expression (this requires that the default element/type namespace is no namespace):

```
17 cast as apple
```

The following shows the use of the constructor function:

```
declare default function namespace ""; apple(17)
```

6 Functions and Operators on Numerics

This section discusses arithmetic operators on the numeric datatypes defined in [XML Schema Part 2: Datatypes Second Edition]. It uses an approach that permits lightweight implementation whenever possible.

6.1 Numeric Types

The operators described in this section are defined on the following numeric types. Each type whose name is indented is derived from the type whose name appears nearest above with one less level of indentation.

xs:decimal	
	xs:integer
xs:float	
xs:double	

They also apply to types derived by restriction from the above types.

Note:

This specification uses [IEEE 754-1985] arithmetic for `xs:float` and `xs:double` values. This differs from [XML Schema Part 2: Datatypes Second Edition] which defines `NaN` as being equal to itself and defines only a single zero in the value space while [IEEE 754-1985] arithmetic treats `NaN` as unequal to all other values including itself and can produce distinct results of positive zero and negative zero. (These are two different machine representations for the same [XML Schema Part 2: Datatypes Second Edition] value.) The text accompanying several functions discusses behaviour for both positive and negative zero inputs and outputs in the interest of alignment with [IEEE 754-1985].

6.2 Operators on Numeric Values

The following functions define the semantics of operators defined in [XQuery 1.0: An XML Query Language] and [XML Path Language (XPath) 2.0] on these numeric types.

Operators	Meaning
`op:numeric-add`	Addition
`op:numeric-subtract`	Subtraction
`op:numeric-multiply`	Multiplication
`op:numeric-divide`	Division
`op:numeric-integer-divide`	Integer division
`op:numeric-mod`	Modulus
`op:numeric-unary-plus`	Unary plus
`op:numeric-unary-minus`	Unary minus (negation)

The parameters and return types for the above operators are the basic numeric types: `xs:integer`, `xs:decimal`, `xs:float` and `xs:double`, and types derived from them. The word "`numeric`" in function signatures signifies these four types. For simplicity, each operator is defined to operate on operands of the same type and return the same type. The exceptions are `op:numeric-divide`, which returns an `xs:decimal` if called with two `xs:integer` operands and `op:numeric-integer-divide` which always returns an `xs:integer`.

If the two operands are not of the same type, *subtype substitution* and *numeric type promotion* are used to obtain two operands of the same type. Section B.1 Type Promotion^{XP} and Section B.2 Operator Mapping^{XP} describe the semantics of these operations in detail.

The result type of operations depends on their argument datatypes and is defined in the following table:

Operator	Returns
`op:operation(xs:integer, xs:integer)`	`xs:integer` (except for `op:numeric-divide(integer, integer)`, which returns `xs:decimal`)
`op:operation(xs:decimal, xs:decimal)`	`xs:decimal`
`op:operation(xs:float, xs:float)`	`xs:float`
`op:operation(xs:double, xs:double)`	`xs:double`
`op:operation(xs:integer)`	`xs:integer`
`op:operation(xs:decimal)`	`xs:decimal`
`op:operation(xs:float)`	`xs:float`
`op:operation(xs:double)`	`xs:double`

These rules define any operation on any pair of arithmetic types. Consider the following example:

```
op:operation(xs:int, xs:double) => op:operation(xs:double, xs:double)
```

For this operation, `xs:int` must be converted to `xs:double`. This can be done, since by the rules above: `xs:int` can be substituted for `xs:integer`, `xs:integer` can be substituted for `xs:decimal`, `xs:decimal` can be promoted to `xs:double`. As far as possible, the promotions should be done in a single step. Specifically, when an `xs:decimal` is promoted to an `xs:double`, it should not be converted to an `xs:float` and then to `xs:double`, as this risks loss of precision.

As another example, a user may define `height` as a derived type of `xs:integer` with a minimum value of 20 and a maximum value of 100. He may then derive `fenceHeight` using an enumeration to restrict the permitted set of values to, say, 36, 48 and 60.

```
op:operation(fenceHeight, xs:integer) => op:operation(xs:integer, xs:integer)
```

`fenceHeight` can be substituted for its base type `height` and `height` can be substituted for its base type `xs:integer`.

On overflow and underflow situations during arithmetic operations conforming implementations ·must· behave as follows:

- For `xs:float` and `xs:double` operations, overflow behavior ·must· be conformant with [IEEE 754-1985]. This specification allows the following options:
  - Raising an error [err:FOAR0002] via an overflow trap.
  - Returning `INF` or `-INF`.
  - Returning the largest (positive or negative) non-infinite number.
- For `xs:float` and `xs:double` operations, underflow behavior ·must· be conformant with [IEEE 754-1985]. This specification allows the following options:
  - Raising an error [err:FOAR0002] via an underflow trap.
  - Returning `0.0E0` or `+/- 2**Emin` or a denormalized value; where `Emin` is the smallest possible `xs:float` or `xs:double` exponent.
- For `xs:decimal` operations, overflow behavior ·must· raise an error [err:FOAR0002]. On underflow, `0.0` must be returned.
- For `xs:integer` operations, implementations that support limited-precision integer operations ·must· select from the following options:
  - They ·may· choose to always raise an error [err:FOAR0002].
  - They ·may· provide an ·implementation-defined· mechanism that allows users to choose between raising an error and returning a result that is modulo the largest representable integer value. See [ISO 10967].

The functions `op:numeric-add`, `op:numeric-subtract`, `op:numeric-multiply`, `op:numeric-divide`, `op:numeric-integer-divide` and `op:numeric-mod` are each defined for pairs of numeric operands, each of which has the same type:`xs:integer`, `xs:decimal`, `xs:float`, or `xs:double`. The functions `op:numeric-unary-plus` and `op:numeric-unary-minus` are defined for a single operand whose type is one of those same numeric types.

For `xs:float` and `xs:double` arguments, if either argument is `NaN`, the result is `NaN`.

For `xs:decimal` values the number of digits of precision returned by the numeric operators is ·implementation-defined·. If the number of digits in the result exceeds the number of digits that the implementation supports, the result is truncated or rounded in an ·implementation-defined· manner.

6.2.1 op:numeric-add

```
op:numeric-add($arg1 as numeric, $arg2 as numeric) as numeric
```

Summary: Backs up the "+" operator and returns the arithmetic sum of its operands: (`$arg1 + $arg2`).

Note:

For `xs:float` or `xs:double` values, if one of the operands is a zero or a finite number and the other is `INF` or `-INF`, `INF` or `-INF` is returned. If both operands are `INF`, `INF` is returned. If both operands are `-INF`, `-INF` is returned. If one of the operands is `INF` and the other is `-INF`, `NaN` is returned.

6.2.2 op:numeric-subtract

```
op:numeric-subtract($arg1 as numeric, $arg2 as numeric) as numeric
```

Summary: Backs up the "-" operator and returns the arithmetic difference of its operands: (`$arg1 - $arg2`).

Note:

For `xs:float` or `xs:double` values, if one of the operands is a zero or a finite number and the other is `INF` or `-INF`, an infinity of the appropriate sign is returned. If both operands are `INF` or `-INF`, `NaN` is returned. If one of the operands is `INF` and the other is `-INF`, an infinity of the appropriate sign is returned.

6.2.3 op:numeric-multiply

```
op:numeric-multiply($arg1 as numeric, $arg2 as numeric) as numeric
```

Summary: Backs up the "*" operator and returns the arithmetic product of its operands: (`$arg1 * $arg2`).

Note:

For `xs:float` or `xs:double` values, if one of the operands is a zero and the other is an infinity, `NaN` is returned. If one of the operands is a non-zero number and the other is an infinity, an infinity with the appropriate sign is returned.

6.2.4 op:numeric-divide

```
op:numeric-divide($arg1 as numeric, $arg2 as numeric) as numeric
```

Summary: Backs up the "div" operator and returns the arithmetic quotient of its operands: (`$arg1 div $arg2`).

As a special case, if the types of both `$arg1` and `$arg2` are `xs:integer`, then the return type is `xs:decimal`.

Notes:

For `xs:decimal` and `xs:integer` operands, if the divisor is (positive or negative) zero, an error is raised [err:FOAR0001]. For `xs:float` and `xs:double` operands, floating point division is performed as specified in [IEEE 754-1985].

For `xs:float` or `xs:double` values, a positive number divided by positive zero returns `INF`. A negative number divided by positive zero returns `-INF`. Division by negative zero returns `-INF` and `INF`, respectively. Positive or negative zero divided by positive or negative zero returns `NaN`. Also, `INF` or `-INF` divided by `INF` or `-INF` returns `NaN`.

6.2.5 op:numeric-integer-divide

`op:numeric-integer-divide($arg1 as numeric, $arg2 as numeric) as xs:integer`

Summary: This function backs up the "idiv" operator by performing an integer division.

If `$arg2` is (positive or negative) zero, then an error is raised [err:FOAR0001]. If either operand is `NaN` or if `$arg1` is `INF` or `-INF` then an error is raised [err:FOAR0002]. If `$arg2` is `INF` or `-INF` (and `$arg1` is not) then the result is zero.

Otherwise, subject to limits of precision and overflow/underflow conditions, the result is the largest (furthest from zero) `xs:integer` value `$N` such that `fn:abs($N * $arg2) le fn:abs($arg1) and fn:compare($N * $arg2, 0) eq fn:compare($arg1, 0)`.

> **Note:**
>
> The second term in this condition ensures that the result has the correct sign.

The implementation may adopt a different algorithm provided that it is equivalent to this formulation in all cases where ·implementation-dependent· or ·implementation-defined· behavior does not affect the outcome, for example, the implementation-defined precision of the result of `xs:decimal` division.

> **Note:**
>
> Except in situations involving errors, loss of precision, or overflow/underflow, the result of `$a idiv $b` is the same as `($a div $b) cast as xs:integer`.

> **Note:**
>
> The semantics of this function are different from integer division as defined in programming languages such as Java and C++.

6.2.5.1 Examples

- `op:numeric-integer-divide(10,3)` returns `3`
- `op:numeric-integer-divide(3,-2)` returns `-1`
- `op:numeric-integer-divide(-3,2)` returns `-1`
- `op:numeric-integer-divide(-3,-2)` returns `1`
- `op:numeric-integer-divide(9.0,3)` returns `3`
- `op:numeric-integer-divide(-3.5,3)` returns `-1`
- `op:numeric-integer-divide(3.0,4)` returns `0`
- `op:numeric-integer-divide(3.1E1,6)` returns `5`
- `op:numeric-integer-divide(3.1E1,7)` returns `4`

6.2.6 op:numeric-mod

`op:numeric-mod($arg1 as numeric, $arg2 as numeric) as numeric`

Summary: Backs up the "mod" operator. Informally, this function returns the remainder resulting from dividing `$arg1`, the dividend, by `$arg2`, the divisor. The operation `a mod b` for operands that are `xs:integer` or `xs:decimal`, or types derived from them, produces a result such that `(a idiv b)*b+(a mod b)` is equal to `a` and the magnitude of the result is always less than the magnitude of `b`. This identity holds even in the special case that the dividend is the negative integer of largest possible magnitude for its type and the divisor is -1 (the remainder is 0). It follows from this rule that the sign of the result is the sign of the dividend.

For `xs:integer` and `xs:decimal` operands, if `$arg2` is zero, then an error is raised [err:FOAR0001].

For `xs:float` and `xs:double` operands the following rules apply:

- If either operand is `NaN`, the result is `NaN`.
- If the dividend is positive or negative infinity, or the divisor is positive or negative zero (0), or both, the result is `NaN`.
- If the dividend is finite and the divisor is an infinity, the result equals the dividend.
- If the dividend is positive or negative zero and the divisor is finite, the result is the same as the dividend.
- In the remaining cases, where neither positive or negative infinity, nor positive or negative zero, nor `NaN` is involved, the result obeys `(a idiv b)*b+(a mod b)` = `a`. Division is truncating division, analogous to integer division, not [IEEE 754-1985] rounding division i.e. additional digits are truncated, not rounded to the required precision.

6.2.6.1 Examples

- `op:numeric-mod(10,3)` returns `1`.
- `op:numeric-mod(6,-2)` returns `0`.
- `op:numeric-mod(4.5,1.2)` returns `0.9`.
- `op:numeric-mod(1.23E2, 0.6E1)` returns `3.0E0`.

6.2.7 op:numeric-unary-plus

```
op:numeric-unary-plus($arg as numeric) as numeric
```

Summary: Backs up the unary "+" operator and returns its operand with the sign unchanged: (+ `$arg`).

The returned value is equal to `$arg`, and is an instance of `xs:integer`, `xs:decimal`, `xs:double`, or `xs:float` depending on the type of `$arg`.

6.2.8 op:numeric-unary-minus

```
op:numeric-unary-minus($arg as numeric) as numeric
```

Summary: Backs up the unary "-" operator and returns its operand with the sign reversed: (- `$arg`).

The returned value is an instance of `xs:integer`, `xs:decimal`, `xs:double`, or `xs:float` depending on the type of `$arg`.

For `xs:integer` and `xs:decimal` arguments, `0` and `0.0` return `0` and `0.0`, respectively. For `xs:float` and `xs:double` arguments, `NaN` returns `NaN`, `0.0E0` returns `-0.0E0` and vice versa. `INF` returns `-INF`. `-INF` returns `INF`.

6.3 Comparison Operators on Numeric Values

This specification defines the following comparison operators on numeric values. Comparisons take two arguments of the same type. If the arguments are of different types, one argument is promoted to the type of the other as described above in 6.2 Operators on Numeric Values. Each comparison operator returns a boolean value. If either, or both, operands are `NaN`, `false` is returned.

Operator	Meaning
`op:numeric-equal`	Equality comparison
`op:numeric-less-than`	Less-than comparison
`op:numeric-greater-than`	Greater-than comparison

6.3.1 op:numeric-equal

```
op:numeric-equal($arg1 as numeric, $arg2 as numeric) as xs:boolean
```

Summary: Returns true if and only if the value of `$arg1` is equal to the value of `$arg2`. For `xs:float` and `xs:double` values, positive zero and negative zero compare equal. `INF` equals `INF` and `-INF` equals `-INF`. `NaN` does not equal itself.

This function backs up the "eq", "ne", "le" and "ge" operators on numeric values.

6.3.2 op:numeric-less-than

```
op:numeric-less-than($arg1 as numeric, $arg2 as numeric) as xs:boolean
```

Summary: Returns `true` if and only if `$arg1` is less than `$arg2`. For `xs:float` and `xs:double` values, positive infinity is greater than all other non-`NaN` values; negative infinity is less than all other non-`NaN` values. If `$arg1` or `$arg2` is `NaN`, the function returns `false`.

This function backs up the "lt" and "le" operators on numeric values.

6.3.3 op:numeric-greater-than

```
op:numeric-greater-than($arg1 as numeric, $arg2 as numeric) as xs:boolean
```

Summary: Returns `true` if and only if `$arg1` is greater than `$arg2`. For `xs:float` and `xs:double` values, positive infinity is greater than all other non-`NaN` values; negative infinity is less than all other non-`NaN` values. If `$arg1` or `$arg2` is `NaN`, the function returns `false`.

This function backs up the "gt" and "ge" operators on numeric values.

6.4 Functions on Numeric Values

The following functions are defined on numeric types. Each function returns a value of the same type as the type of its argument.

- If the argument is the empty sequence, the empty sequence is returned.
- For `xs:float` and `xs:double` arguments, if the argument is "NaN", "NaN" is returned.
- Except for `fn:abs()`, for `xs:float` and `xs:double` arguments, if the argument is positive or negative infinity, positive or negative infinity is returned.

Function	Meaning
`fn:abs`	Returns the absolute value of the argument.

Function	Meaning
fn:ceiling	Returns the smallest number with no fractional part that is greater than or equal to the argument.
fn:floor	Returns the largest number with no fractional part that is less than or equal to the argument.
fn:round	Rounds to the nearest number with no fractional part.
fn:round-half-to-even	Takes a number and a precision and returns a number rounded to the given precision. If the fractional part is exactly half, the result is the number whose least significant digit is even.

6.4.1 fn:abs

```
fn:abs($arg as numeric?) as numeric?
```

Summary: Returns the absolute value of `$arg`. If `$arg` is negative returns `-$arg` otherwise returns `$arg`. If type of `$arg` is one of the four numeric types `xs:float`, `xs:double`, `xs:decimal` or `xs:integer` the type of the result is the same as the type of `$arg`. If the type of `$arg` is a type derived from one of the numeric types, the result is an instance of the base numeric type.

For `xs:float` and `xs:double` arguments, if the argument is positive zero or negative zero, then positive zero is returned. If the argument is positive or negative infinity, positive infinity is returned.

For detailed type semantics, see Section 7.2.3 The fn:abs, fn:ceiling, fn:floor, fn:round, and fn:round-half-to-even functions*FS*

6.4.1.1 Examples

- `fn:abs(10.5)` returns `10.5`.
- `fn:abs(-10.5)` returns `10.5`.

6.4.2 fn:ceiling

```
fn:ceiling($arg as numeric?) as numeric?
```

Summary: Returns the smallest (closest to negative infinity) number with no fractional part that is not less than the value of `$arg`. If type of `$arg` is one of the four numeric types `xs:float`, `xs:double`, `xs:decimal` or `xs:integer` the type of the result is the same as the type of `$arg`. If the type of `$arg` is a type derived from one of the numeric types, the result is an instance of the base numeric type.

For `xs:float` and `xs:double` arguments, if the argument is positive zero, then positive zero is returned. If the argument is negative zero, then negative zero is returned. If the argument is less than zero and greater than -1, negative zero is returned.

For detailed type semantics, see Section 7.2.3 The fn:abs, fn:ceiling, fn:floor, fn:round, and fn:round-half-to-even functions*FS*

6.4.2.1 Examples

- `fn:ceiling(10.5)` returns `11`.
- `fn:ceiling(-10.5)` returns `-10`.

6.4.3 fn:floor

```
fn:floor($arg as numeric?) as numeric?
```

Summary: Returns the largest (closest to positive infinity) number with no fractional part that is not greater than the value of `$arg`. If type of `$arg` is one of the four numeric types `xs:float`, `xs:double`, `xs:decimal` or `xs:integer` the type of the result is the same as the type of `$arg`. If the type of `$arg` is a type derived from one of the numeric types, the result is an instance of the base numeric type.

For `float` and `double` arguments, if the argument is positive zero, then positive zero is returned. If the argument is negative zero, then negative zero is returned.

For detailed type semantics, see Section 7.2.3 The fn:abs, fn:ceiling, fn:floor, fn:round, and fn:round-half-to-even functions*FS*

6.4.3.1 Examples

- `fn:floor(10.5)` returns `10`.
- `fn:floor(-10.5)` returns `-11`.

6.4.4 fn:round

```
fn:round($arg as numeric?) as numeric?
```

Summary: Returns the number with no fractional part that is closest to the argument. If there are two such numbers, then the one that is closest to positive infinity is returned. If type of `$arg` is one of the four numeric types `xs:float`, `xs:double`, `xs:decimal` or `xs:integer` the type of the result is the same as the type of `$arg`. If the type of `$arg` is a type derived from one of the numeric types, the result is an instance of the base numeric type.

For `xs:float` and `xs:double` arguments, if the argument is positive infinity, then positive infinity is returned. If the argument is negative infinity, then negative infinity is returned. If the argument is positive zero, then positive zero is returned. If the argument is negative zero, then negative zero is returned. If the argument is less than zero, but greater than or equal to -0.5, then negative zero is returned. In the cases where positive zero or negative zero is returned, negative zero or positive zero may be returned as [XML Schema Part 2: Datatypes Second Edition] does not distinguish between the values positive zero and negative zero.

For the last two cases, note that the result is not the same as `fn:floor(x+0.5)`.

For detailed type semantics, see Section 7.2.3 The fn:abs, fn:ceiling, fn:floor, fn:round, and fn:round-half-to-even functions*FS*

6.4.4.1 Examples

- `fn:round(2.5)` returns `3.`
- `fn:round(2.4999)` returns `2`.
- `fn:round(-2.5)` returns `-2` (not the possible alternative, `-3`).

6.4.5 fn:round-half-to-even

```
fn:round-half-to-even($arg as numeric?) as numeric?
```

```
fn:round-half-to-even($arg as numeric?, $precision as xs:integer) as numeric?
```

Summary: The value returned is the nearest (that is, numerically closest) value to `$arg` that is a multiple of ten to the power of minus `$precision`. If two such values are equally near (e.g. if the fractional part in `$arg` is exactly .500...), the function returns the one whose least significant digit is even.

If the type of `$arg` is one of the four numeric types `xs:float`, `xs:double`, `xs:decimal` or `xs:integer` the type of the result is the same as the type of `$arg`. If the type of `$arg` is a type derived from one of the numeric types, the result is an instance of the base numeric type.

The first signature of this function produces the same result as the second signature with `$precision=0`.

For arguments of type `xs:float` and `xs:double`, if the argument is `NaN`, positive or negative zero, or positive or negative infinity, then the result is the same as the argument. In all other cases, the argument is cast to `xs:decimal`, the function is applied to this `xs:decimal` value, and the resulting `xs:decimal` is cast back to `xs:float` or `xs:double` as appropriate to form the function result. If the resulting `xs:decimal` value is zero, then positive or negative zero is returned according to the sign of the original argument.

Note that the process of casting to `xs:decimal` may result in an error [err:FOCA0001].

If `$arg` is of type `xs:float` or `xs:double`, rounding occurs on the value of the mantissa computed with exponent = 0.

For detailed type semantics, see Section 7.2.3 The fn:abs, fn:ceiling, fn:floor, fn:round, and fn:round-half-to-even functions*FS*

> **Note:**
>
> This function is typically used in financial applications where the argument is of type `xs:decimal`. For arguments of type `xs:float` and `xs:double` the results may be counterintuitive. For example, consider `round-half-to-even(xs:float(150.0150), 2)`.
>
> An implementation that supports 18 digits for `xs:decimal` will convert the argument to the `xs:decimal` 150.014999389... which will then be rounded to the `xs:decimal` 150.01 which will be converted back to the `xs:float` whose exact value is 150.0099945068... whereas `round-half-to-even(xs:decimal(150.0150), 2)` will result in the `xs:decimal` whose exact value is 150.02.

6.4.5.1 Examples

- `fn:round-half-to-even(0.5)` returns `0`.
- `fn:round-half-to-even(1.5)` returns `2`.
- `fn:round-half-to-even(2.5)` returns `2`.
- `fn:round-half-to-even(3.567812E+3, 2)` returns `3567.81E0`.
- `fn:round-half-to-even(4.7564E-3, 2)` returns `0.0E0`.
- `fn:round-half-to-even(35612.25, -2)` returns `35600`.

7 Functions on Strings

This section discusses functions and operators on the [XML Schema Part 2: Datatypes Second Edition] `xs:string` datatype and the datatypes derived from it.

7.1 String Types

The operators described in this section are defined on the following types. Each type whose name is indented is derived from the type whose name appears nearest above with one less level of indentation.

xs:string					
	xs:normalizedString				
		xs:token			
			xs:language		
			xs:NMTOKEN		
			xs:Name		
				xs:NCName	
					xs:ID
					xs:IDREF
					xs:ENTITY

They also apply to user-defined types derived by restriction from the above types.

It is ·implementation-defined· which version of [The Unicode Standard] is supported, but it is recommended that the most recent version of Unicode be used.

Unless explicitly stated, the `xs:string` values returned by the functions in this document are not normalized in the sense of [Character Model for the World Wide Web 1.0: Fundamentals].

Notes:

This document uses the term "code point", sometimes spelt "codepoint" (also known as "character number" or "code position") to mean a non-negative integer that represents a character in some encoding. See [Character Model for the World Wide Web 1.0: Fundamentals]. The use of the word "character" in this document is in the sense of production [2] of [Extensible Markup Language (XML) 1.0 Recommendation (Third Edition)]. [The Unicode Standard], defines code points that range from #x0000 to #x10FFFF inclusive and may include code points that have not yet been assigned to characters.

In functions that involve character counting such as `fn:substring`, `fn:string-length` and `fn:translate`, what is counted is the number of XML characters in the string (or equivalently, the number of Unicode code points). Some implementations may represent a code point above xFFFF using two 16-bit values known as a surrogate. A surrogate counts as one character, not two.

7.2 Functions to Assemble and Disassemble Strings

Function	Meaning
`fn:codepoints-to-string`	Creates an `xs:string` from a sequence of Unicode code points.
`fn:string-to-codepoints`	Returns the sequence of Unicode code points that constitute an `xs:string`.

7.2.1 fn:codepoints-to-string

```
fn:codepoints-to-string($arg as xs:integer*) as xs:string
```

Summary: Creates an `xs:string` from a sequence of [The Unicode Standard] code points. Returns the zero-length string if `$arg` is the empty sequence. If any of the code points in `$arg` is not a legal XML character, an error is raised [err:FOCH0001].

7.2.1.1 Examples

- `fn:codepoints-to-string((2309, 2358, 2378, 2325))` returns "अशॊक"

7.2.2 fn:string-to-codepoints

```
fn:string-to-codepoints($arg as xs:string?) as xs:integer*
```

Summary: Returns the sequence of [The Unicode Standard] code points that constitute an `xs:string`. If `$arg` is a zero-length string or the empty sequence, the empty sequence is returned.

7.2.2.1 Examples

- `fn:string-to-codepoints("Thérèse")` returns the sequence (84, 104, 233, 114, 232, 115, 101)

7.3 Equality and Comparison of Strings

7.3.1 Collations

A collation is a specification of the manner in which character strings are compared and, by extension, ordered. When values whose type is `xs:string` or a type derived from `xs:string` are compared (or, equivalently, sorted), the comparisons are inherently performed according to some collation (even if that collation is defined entirely on code point values). The [Character Model for the World Wide Web 1.0: Fundamentals] observes that some applications may require different comparison and ordering behaviors than other applications. Similarly, some users having particular linguistic expectations may require different behaviors than other users. Consequently, the collation must be taken into account when comparing strings in any context. Several functions in this and the following section make use of a collation.

Collations can indicate that two different code points are, in fact, equal for comparison purposes (e.g., "v" and "w" are considered equivalent in Swedish). Strings can be compared codepoint-by-codepoint or in a linguistically appropriate manner, as defined by the collation.

Some collations, especially those based on the [Unicode Collation Algorithm] can be "tailored" for various purposes. This document does not discuss such tailoring, nor does it provide a mechanism to perform tailoring. Instead, it assumes that the collation argument to the various functions below is a tailored and named collation. A specific collation with a distinguished name, `http://www.w3.org/2005/xpath-functions/collation/codepoint`, provides the ability to compare strings based on code point values. Every implementation of XQuery/XPath must support the collation based on code point values.

In the ideal case, a collation should treat two strings as equal if the two strings are identical after Unicode normalization. Thus, the [Character Model for the World Wide Web 1.0: Normalization] recommends that all strings be subjected to early Unicode normalization and some collations will raise runtime errors if they encounter strings that are not properly normalized. However, it is not possible to guarantee that all strings in all XML documents are, in fact, normalized, or that they are normalized in the same manner. In order to maximize interoperability of operations on XML documents in general, there may be collations that operate on unnormalized strings and other collations that implicitly normalize strings before comparing them. Applications may choose the kind of collation best suited for their needs. Note that collations based on the Unicode collation algorithm implicitly normalize strings before comparison and produce equivalent results regardless of a string's normalization.

This specification assumes that collations are named and that the collation name may be provided as an argument to string functions. Functions that allow specification of a collation do so with an argument whose type is `xs:string` but whose lexical form must conform to an `xs:anyURI`. If the collation is specified using a relative URI, it is assumed to be relative to the value of the base-uri property in the static context. This specification also defines the manner in which a default collation is determined if the collation argument is not specified in invocations of functions that use a collation but allow it to be omitted.

This specification does not define whether or not the collation URI is dereferenced. The collation URI may be an abstract identifier, or it may refer to an actual resource describing the collation. If it refers to a resource, this specification does not define the nature of that resource. One possible candidate is that the resource is a locale description expressed using the Locale Data Markup Language: see [Locale Data Markup Language].

Functions such as `fn:compare` and `fn:max` that compare `xs:string` values use a single collation URI to identify all aspects of the collation rules. This means that any parameters such as the strength of the collation must be specified as part of the collation URI. For example, suppose there is a collation " `http://www.example.com/collations/French` " that refers to a French collation that compares on the basis of base characters. Collations that use the same basic rules, but with higher strengths, for example, base characters and accents, or base characters, accents and case, would need to be given different names, say " `http://www.example.com/collations/French1` " and " `http://www.example.com/collations/French2` ". Note that some specifications use the term collation to refer to an algorithm that can be parameterized, but in this specification, each possible parameterization is considered to be a distinct collation.

The XQuery/XPath static context includes a provision for a default collation that can be used for string comparisons and ordering operations. See the description of the static context in Section 2.1.1 Static Context^XP. If the default collation is not specified by the user or the system, the default collation is the Unicode code point collation (`http://www.w3.org/2005/xpath-functions/collation/codepoint`).

The decision of which collation to use for a given comparison or ordering function is determined by the following algorithm:

1. If the function specifies an explicit collation, CollationA (e.g., if the optional collation argument is specified in an invocation of the `fn:compare()` function), then:
   - If CollationA is supported by the implementation, then CollationA is used.
   - Otherwise, an error is raised [err:FOCH0002].
2. If no collation is explicitly specified for the function and the default collation in the XQuery/XPath static context is CollationB, then:
   - If CollationB is supported by the implementation, then CollationB is used.
   - Otherwise, an error is raised [err:FOCH0002].

Note:

XML allows elements to specify the `xml:lang` attribute to indicate the language associated with the content of such an element. This specification does not use `xml:lang` to identify the default collation because using `xml:lang` does not produce desired effects when the two strings to be compared have different `xml:lang` values or when a string is multilingual.

Function	Meaning

fn:compare	Returns -1, 0, or 1, depending on whether the value of the first argument is respectively less than, equal to, or greater than the value of the second argument, according to the rules of the collation that is used.
fn:codepoint-equal	Returns `true` if the two arguments are equal using the Unicode code point collation.

7.3.2 fn:compare

```
fn:compare($comparand1 as xs:string?, $comparand2 as xs:string?) as xs:integer?
fn:compare($comparand1 as xs:string?,
           $comparand2 as xs:string?,
           $collation  as xs:string) as xs:integer?
```

Summary: Returns -1, 0, or 1, depending on whether the value of the `$comparand1` is respectively less than, equal to, or greater than the value of `$comparand2`, according to the rules of the collation that is used.

The collation used by the invocation of this function is determined according to the rules in **7.3.1 Collations**.

If either argument is the empty sequence, the result is the empty sequence.

This function, invoked with the first signature, backs up the "eq", "ne", "gt", "lt", "le" and "ge" operators on string values.

7.3.2.1 Examples

- `fn:compare('abc', 'abc')` returns 0.
- `fn:compare('Strasse', 'Straße')` returns 0 if and only if the default collation includes provisions that equate "ss" and the (German) character "ß" ("sharp-s"). (Otherwise, the returned value depends on the semantics of the default collation.)
- `fn:compare('Strasse', 'Straße', 'deutsch')` returns 0 if the collation identified by the relative URI constructed from the `string` value "deutsch" includes provisions that equate "ss" and the (German) character "ß" ("sharp-s"). (Otherwise, the returned value depends on the semantics of that collation.)
- `fn:compare('Strassen', 'Straße')` returns 1 if the default collation includes provisions that treat differences between "ss" and the (German) character "ß" ("sharp-s") with less strength than the differences between the base characters, such as the final "n".

7.3.3 fn:codepoint-equal

```
fn:codepoint-equal($comparand1 as xs:string?,
                   $comparand2 as xs:string?) as xs:boolean?
```

Summary: Returns `true` or `false` depending on whether the value of `$comparand1` is equal to the value of `$comparand2`, according to the Unicode code point collation (`http://www.w3.org/2005/xpath-functions/collation/codepoint`).

If either argument is the empty sequence, the result is the empty sequence.

Note:

This function allows xs:anyURI values to be compared without having to specify the Unicode code point collation.

7.4 Functions on String Values

The following functions are defined on values of type xs:string and types derived from it.

Function	Meaning
fn:concat	Concatenates two or more xs:anyAtomicType arguments cast to xs:string.
fn:string-join	Returns the xs:string produced by concatenating a sequence of xs:strings using an optional separator.
fn:substring	Returns the xs:string located at a specified place within an argument xs:string.
fn:string-length	Returns the length of the argument.
fn:normalize-space	Returns the whitespace-normalized value of the argument.
fn:normalize-unicode	Returns the normalized value of the first argument in the normalization form specified by the second argument.
fn:upper-case	Returns the upper-cased value of the argument.
fn:lower-case	Returns the lower-cased value of the argument.
fn:translate	Returns the first xs:string argument with occurrences of characters contained in the second argument replaced by the character at the corresponding position in the third argument.
fn:encode-for-uri	Returns the xs:string argument with certain characters escaped to enable the resulting string to be used as a path segment in a URI.
fn:iri-to-uri	Returns the xs:string argument with certain characters escaped to enable the resulting string to be used as (part of) a URI.
fn:escape-html-uri	Returns the xs:string argument with certain characters escaped in the manner that html user agents handle attribute values that expect URIs.

Notes:

When the above operators and functions are applied to datatypes derived from xs:string, they are guaranteed to return legal xs:strings, but they might not return a legal value for the particular subtype to which they were applied.

The strings returned by fn:concat and fn:string-join are not guaranteed to be normalized. But see note in fn:concat.

7.4.1 fn:concat

```
fn:concat($arg1 as xs:anyAtomicType?,
          $arg2 as xs:anyAtomicType?,
          ...   ) as xs:string
```

Summary: Accepts two or more xs:anyAtomicType arguments and casts them to xs:string. Returns the xs:string that is the concatenation of the values of its arguments after conversion. If any of the arguments is the empty sequence, the argument is treated as the zero-length string.

The fn:concat function is specified to allow two or more arguments, which are concatenated together. This is the only function specified in this document that allows a variable number of arguments. This capability is retained for compatibility with [XML Path Language (XPath) Version 1.0].

Note:

As mentioned in **7.1 String Types** Unicode normalization is not automatically applied to the result of `fn:concat`. If a normalized result is required, `fn:normalize-unicode` can be applied to the `xs:string` returned by `fn:concat`. The following XQuery:

```
let $v1 := "I plan to go to Mu"
let $v2 := "?nchen in September"
return concat($v1, $v2)
```

where the "?" represents either the actual Unicode character COMBINING DIARESIS (Unicode codepoint U+0308) or "&#x0308;", will return:

"I plan to go to Mu?nchen in September"

where the "?" represents either the actual Unicode character COMBINING DIARESIS (Unicode codepoint U+0308) or "&#x0308;". It is worth noting that the returned value is not normalized in NFC; however, it is normalized in NFD. .

However, the following XQuery:

```
let $v1 := "I plan to go to Mu"
let $v2 := "?nchen in September"
return normalize-unicode(concat($v1, $v2))
```

where the "?" represents either the actual Unicode character COMBINING DIARESIS (Unicode codepoint U+0308) or "&#x0308;", will return:

"I plan to go to München in September"

This returned result is normalized in NFC.

7.4.1.1 Examples

- `fn:concat('un', 'grateful')` returns `"ungrateful"`.
- `fn:concat('Thy ', (), 'old ', "groans", "", ' ring', ' yet', ' in', ' my', ' ancient',' ears.')` returns `"Thy old groans ring yet in my ancient ears."`.
- `fn:concat('Ciao!',())` returns `"Ciao!"`.
- `fn:concat('Ingratitude, ', 'thou ', 'marble-hearted', ' fiend!')` returns `"Ingratitude, thou marble-hearted fiend!"`.

7.4.2 fn:string-join

```
fn:string-join($arg1 as xs:string*, $arg2 as xs:string) as xs:string
```

Summary: Returns a `xs:string` created by concatenating the members of the `$arg1` sequence using `$arg2` as a separator. If the value of `$arg2` is the zero-length string, then the members of `$arg1` are concatenated without a separator.

If the value of `$arg1` is the empty sequence, the zero-length string is returned.

7.4.2.1 Examples

- `fn:string-join(('Now', 'is', 'the', 'time', '...'), ' ')` returns `"Now is the time ..."`.
- `fn:string-join(('Blow, ', 'blow, ', 'thou ', 'winter ', 'wind!'), '')` returns `"Blow, blow, thou winter wind!"`.
- `fn:string-join((), 'separator')` returns `""`.
- Assume a document:

  ```
  <doc>
    <chap>
      <section>
      </section>
    </chap>
  </doc>
  ```

  with the `<section>` as the context node, the [XML Path Language (XPath) 2.0] expression:

  `fn:string-join(for $n in ancestor-or-self::* return name($n), '/')`

  returns `"doc/chap/section"`

7.4.3 fn:substring

```
fn:substring($sourceString as xs:string?,
             $startingLoc  as xs:double) as xs:string
fn:substring($sourceString as xs:string?,
             $startingLoc  as xs:double,
             $length       as xs:double) as xs:string
```

Summary: Returns the portion of the value of `$sourceString` beginning at the position indicated by the value of `$startingLoc` and continuing for the number of characters indicated by the value of `$length`. The characters returned do not extend beyond `$sourceString`. If `$startingLoc` is zero or negative, only those characters in positions greater than zero are returned.

More specifically, the three argument version of the function returns the characters in `$sourceString` whose position `$p` obeys:

`fn:round($startingLoc) <= $p < fn:round($startingLoc) + fn:round($length)`

The two argument version of the function assumes that `$length` is infinite and returns the characters in `$sourceString` whose position `$p` obeys:

`fn:round($startingLoc) <= $p < fn:round(INF)`

In the above computations, the rules for `op:numeric-less-than()` and `op:numeric-greater-than()` apply.

If the value of `$sourceString` is the empty sequence, the zero-length string is returned.

> **Note:**
>
> The first character of a string is located at position 1, not position 0.

7.4.3.1 Examples

- `fn:substring("motor car", 6)` returns `" car"`.

  Characters starting at position 6 to the end of `$sourceString` are selected.

- `fn:substring("metadata", 4, 3)` returns `"ada"`.

  Characters at positions greater than or equal to 4 and less than 7 are selected.
- `fn:substring("12345", 1.5, 2.6)` returns `"234"`.

  Characters at positions greater than or equal to 2 and less than 5 are selected.
- `fn:substring("12345", 0, 3)` returns `"12"`.

  Characters at positions greater than or equal to 0 and less than 3 are selected. Since the first position is 1, these are the characters at positions 1 and 2.
- `fn:substring("12345", 5, -3)` returns `""`.

  Characters at positions greater than or equal to 5 and less than 2 are selected.
- `fn:substring("12345", -3, 5)` returns `"1"`.

  Characters at positions greater than or equal to -3 and less than 2 are selected. Since the first position is 1, this is the character at position 1.
- `fn:substring("12345", 0 div 0E0, 3)` returns `""`.

  Since `0 div 0E0` returns `NaN`, and `NaN` compared to any other number returns `false`, no characters are selected.
- `fn:substring("12345", 1, 0 div 0E0)` returns `""`.

  As above.
- `fn:substring((), 1, 3)` returns `""`.
- `fn:substring("12345", -42, 1 div 0E0)` returns `"12345"`.

  Characters at positions greater than or equal to -42 and less than INF are selected.
- `fn:substring("12345", -1 div 0E0, 1 div 0E0)` returns `""`.

  Since `-INF + INF` returns `NaN`, no characters are selected.

7.4.4 fn:string-length

```
fn:string-length() as xs:integer
```

```
fn:string-length($arg as xs:string?) as xs:integer
```

Summary: Returns an `xs:integer` equal to the length in characters of the value of `$arg`.

If the value of `$arg` is the empty sequence, the `xs:integer` 0 is returned.

If no argument is supplied, `$arg` defaults to the string value (calculated using `fn:string()`) of the context item (`.`). If no argument is supplied and the context item is undefined an error is raised: [err:XPDY0002]^XP.

7.4.4.1 Examples

- `fn:string-length("Harp not on that string, madam; that is past.")` returns `45`.
- `fn:string-length(())` returns `0`.

7.4.5 fn:normalize-space

```
fn:normalize-space() as xs:string
```

```
fn:normalize-space($arg as xs:string?) as xs:string
```

Summary: Returns the value of `$arg` with whitespace normalized by stripping leading and trailing whitespace and replacing sequences of one or more than one whitespace character with a single space, `#x20`.

The whitespace characters are defined in the metasymbol S (Production 3) of [Extensible Markup Language (XML) 1.0 Recommendation (Third Edition)].

Note:

The definition of the metasymbol S (Production 3), is unchanged in [Extensible Markup Language (XML) 1.1 Recommendation].

If the value of `$arg` is the empty sequence, returns the zero-length string.

If no argument is supplied, then `$arg` defaults to the string value (calculated using `fn:string()`) of the context item (`.`). If no argument is supplied and the context item is undefined an error is raised: [err:XPDY0002]XP.

7.4.5.1 Examples

- `fn:normalize-space(" The  wealthy curled darlings of   our  nation. ")` returns `"The wealthy curled darlings of our nation."`.
- `fn:normalize-space(())` returns "".

7.4.6 fn:normalize-unicode

```
fn:normalize-unicode($arg as xs:string?) as xs:string
fn:normalize-unicode($arg               as xs:string?,
                     $normalizationForm as xs:string) as xs:string
```

Summary: Returns the value of `$arg` normalized according to the normalization criteria for a normalization form identified by the value of `$normalizationForm`. The effective value of the `$normalizationForm` is computed by removing leading and trailing blanks, if present, and converting to upper case.

If the value of `$arg` is the empty sequence, returns the zero-length string.

See [Character Model for the World Wide Web 1.0: Normalization] for a description of the normalization forms.

If the `$normalizationForm` is absent, as in the first format above, it shall be assumed to be "NFC"

- If the effective value of `$normalizationForm` is "NFC", then the value returned by the function is the value of `$arg` in Unicode Normalization Form C (NFC).
- If the effective value of `$normalizationForm` is "NFD", then the value returned by the function is the value of `$arg` in Unicode Normalization Form D (NFD).
- If the effective value of `$normalizationForm` is "NFKC", then the value returned by the function is the value of `$arg` in Unicode Normalization Form KC (NFKC).
- If the effective value of `$normalizationForm` is "NFKD", then the value returned by the function is the value of `$arg` in Unicode Normalization Form KD (NFKD).
- If the effective value of `$normalizationForm` is "FULLY-NORMALIZED", then the value returned by the function is the value of `$arg` in the fully normalized form.
- If the effective value of `$normalizationForm` is the zero-length string, no normalization is performed and `$arg` is returned.

Conforming implementations ·must· support normalization form "NFC" and ·may· support normalization forms "NFD", "NFKC", "NFKD", "FULLY-NORMALIZED". They ·may· also support other normalization forms with ·implementation-defined· semantics. If the effective value of the `$normalizationForm` is other than one of the values supported by the implementation, then an error is raised [err:FOCH0003].

7.4.7 fn:upper-case

```
fn:upper-case($arg as xs:string?) as xs:string
```

Summary: Returns the value of `$arg` after translating every character to its upper-case correspondent as defined in the appropriate case mappings section in the Unicode standard [The Unicode Standard]. For versions of Unicode beginning with the 2.1.8 update, only locale-insensitive case mappings should be applied. Beginning with version 3.2.0 (and likely future versions) of Unicode, precise mappings are described in default case operations, which are full case mappings in the absence of tailoring for particular languages and environments. Every lower-case character that does not have an upper-case correspondent, as well as every upper-case character, is included in the returned value in its original form.

If the value of `$arg` is the empty sequence, the zero-length string is returned.

> **Note:**
>
> Case mappings may change the length of a string. In general, the two functions are not inverses of each other `fn:lower-case(fn:upper-case($arg))` is not guaranteed to return `$arg`, nor is `fn:upper-case(fn:lower-case($arg))`. The Latin small letter dotless i (as used in Turkish) is perhaps the most prominent lower-case letter which will not round-trip. The Latin capital letter i with dot above is the most prominent upper-case letter which will not round trip; there are others.
>
> These functions may not always be linguistically appropriate (e.g. Turkish i without dot) or appropriate for the application (e.g. titlecase). In cases such as Turkish, a simple translation should be used first.
>
> Results may violate user expectations (in Quebec, for example, the standard uppercase equivalent of "è" is "È", while in metropolitan France it is more commonly "E"; only one of these is supported by the functions as defined).
>
> Many characters of class Ll lack uppercase equivalents in the Unicode case mapping tables; many characters of class Lu lack lowercase equivalents.

7.4.7.1 Examples

- `fn:upper-case("abCd0")` returns `"ABCD0"`.

7.4.8 fn:lower-case

```
fn:lower-case($arg as xs:string?) as xs:string
```

Summary: Returns the value of `$arg` after translating every character to its lower-case correspondent as defined in the appropriate case mappings section in the Unicode standard [The Unicode Standard]. For versions of Unicode beginning with the 2.1.8 update, only locale-insensitive case mappings should be applied. Beginning with version 3.2.0 (and likely future versions) of Unicode, precise mappings are described in default case operations, which are full case mappings in the absence of tailoring for particular languages and environments. Every upper-case character that does not have a lower-case correspondent, as well as every lower-case character, is included in the returned value in its original form.

If the value of `$arg` is the empty sequence, the zero-length string is returned.

> **Note:**
>
> Case mappings may change the length of a string. In general, the two functions are not inverses of each other `fn:lower-case(fn:upper-case($arg))` is not guaranteed to return `$arg`, nor is `fn:upper-case(fn:lower-case($arg))`. The Latin small letter dotless i (as used in Turkish) is perhaps the most prominent lower-case letter which will not round-trip. The Latin capital letter i with dot above is the most prominent upper-case letter which will not round trip; there are others.
>
> These functions may not always be linguistically appropriate (e.g. Turkish i without dot) or appropriate for the application (e.g. titlecase). In cases such as Turkish, a simple translation should be used first.
>
> Results may violate user expectations (in Quebec, for example, the standard uppercase equivalent of "è" is "È", while in metropolitan France it is more commonly "E"; only one of these is supported by the functions as defined).
>
> Many characters of class Ll lack uppercase equivalents in the Unicode case mapping tables; many characters of class Lu lack lowercase equivalents.

7.4.8.1 Examples

- `fn:lower-case("ABc!D")` returns `"abc!d"`.

7.4.9 fn:translate

```
fn:translate($arg         as xs:string?,
             $mapString   as xs:string,
             $transString as xs:string) as xs:string
```

Summary: Returns the value of `$arg` modified so that every character in the value of `$arg` that occurs at some position *N* in the value of `$mapString` has been replaced by the character that occurs at position *N* in the value of `$transString`.

If the value of `$arg` is the empty sequence, the zero-length string is returned.

Every character in the value of `$arg` that does not appear in the value of `$mapString` is unchanged.

Every character in the value of `$arg` that appears at some position *M* in the value of `$mapString`, where the value of `$transString` is less than *M* characters in length, is omitted from the returned value. If `$mapString` is the zero-length string `$arg` is returned.

If a character occurs more than once in `$mapString`, then the first occurrence determines the replacement character. If `$transString` is longer than `$mapString`, the excess characters are ignored.

7.4.9.1 Examples

- `fn:translate("bar","abc","ABC")` returns `"BAr"`

- `fn:translate("--aaa--","abc-","ABC")` returns `"AAA"`.
- `fn:translate("abcdabc", "abc", "AB")` returns `"ABdAB"`.

7.4.10 fn:encode-for-uri

```
fn:encode-for-uri($uri-part as xs:string?) as xs:string
```

Summary: This function encodes reserved characters in an `xs:string` that is intended to be used in the path segment of a URI. It is invertible but not idempotent. This function applies the URI escaping rules defined in section 2 of [RFC 3986] to the `xs:string` supplied as `$uri-part`. The effect of the function is to escape reserved characters. Each such character in the string is replaced with its percent-encoded form as described in [RFC 3986].

If `$uri-part` is the empty sequence, returns the zero-length string.

All characters are escaped except those identified as "unreserved" by [RFC 3986], that is the upper- and lower-case letters A-Z, the digits 0-9, HYPHEN-MINUS ("-"), LOW LINE ("_"), FULL STOP ".", and TILDE "~".

Note that this function escapes URI delimiters and therefore cannot be used indiscriminately to encode "invalid" characters in a path segment.

Since [RFC 3986] recommends that, for consistency, URI producers and normalizers should use uppercase hexadecimal digits for all percent-encodings, this function must always generate hexadecimal values using the upper-case letters A-F.

7.4.10.1 Examples

- `fn:encode-for-uri("http://www.example.com/00/Weather/CA/Los%20Angeles#ocean")` returns `"http%3A%2F%2Fwww.example.com%2F00%2FWeather%2FCA%2FLos%2520Angeles%23ocean"`. This is probably not what the user intended because all of the delimiters have been encoded.
- `concat("http://www.example.com/", encode-for-uri("~bébé"))` returns `"http://www.example.com/~b%C3%A9b%C3%A9"`.
- `concat("http://www.example.com/", encode-for-uri("100% organic"))` returns `"http://www.example.com/100%25%20organic"`.

7.4.11 fn:iri-to-uri

```
fn:iri-to-uri($iri as xs:string?) as xs:string
```

Summary: This function converts an `xs:string` containing an IRI into a URI according to the rules spelled out in Section 3.1 of [RFC 3987]. It is idempotent but not invertible.

If `$iri` contains a character that is invalid in an IRI, such as the space character (see note below), the invalid character is replaced by its percent-encoded form as described in [RFC 3986] before the conversion is performed.

If `$iri` is the empty sequence, returns the zero-length string.

Since [RFC 3986] recommends that, for consistency, URI producers and normalizers should use uppercase hexadecimal digits for all percent-encodings, this function must always generate hexadecimal values using the upper-case letters A-F.

Notes:

This function does not check whether `$iri` is a legal IRI. It treats it as an `xs:string` and operates on the characters in the `xs:string`.

The following printable ASCII characters are invalid in an IRI: "<", ">", " " " (double quote), space, "{", "}", "|", "\", "^", and "`". Since these characters should not appear in an IRI, if they do appear in `$iri` they will be percent-encoded. In addition, characters outside the range x20-x7E will be percent-encoded because they are invalid in a URI.

Since this function does not escape the PERCENT SIGN "%" and this character is not allowed in data within a URI, users wishing to convert character strings, such as file names, that include "%" to a URI should manually escape "%" by replacing it with "%25".

7.4.11.1 Examples

- `fn:iri-to-uri ("http://www.example.com/00/Weather/CA/Los%20Angeles#ocean")` returns `"http://www.example.com/00/Weather/CA/Los%20Angeles#ocean"`.
- `fn:iri-to-uri ("http://www.example.com/~bébé")` returns `"http://www.example.com/~b%C3%A9b%C3%A9"`.

7.4.12 fn:escape-html-uri

`fn:escape-html-uri`(`$uri as` *`xs:string?`*) `as` *`xs:string`*

Summary: This function escapes all characters except printable characters of the US-ASCII coded character set, specifically the octets ranging from 32 to 126 (decimal). The effect of the function is to escape a URI in the manner html user agents handle attribute values that expect URIs. Each character in `$uri` to be escaped is replaced by an escape sequence, which is formed by encoding the character as a sequence of octets in UTF-8, and then representing each of these octets in the form %HH, where HH is the hexadecimal representation of the octet. This function must always generate hexadecimal values using the upper-case letters A-F.

If `$uri` is the empty sequence, returns the zero-length string.

Note:

The behavior of this function corresponds to the recommended handling of non-ASCII characters in URI attribute values as described in [HTML 4.0] Appendix B.2.1.

7.4.12.1 Examples

- `fn:escape-html-uri ("http://www.example.com/00/Weather/CA/Los Angeles#ocean")` returns `"http://www.example.com/00/Weather/CA/Los Angeles#ocean"`.
- `fn:escape-html-uri ("javascript:if (navigator.browserLanguage == 'fr') window.open('http://www.example.com/~bébé');") returns "javascript:if (navigator.browserLanguage == 'fr') window.open('http://www.example.com/~b%C3%A9b%C3%A9');"`.

7.5 Functions Based on Substring Matching

The functions described in the section examine a string `$arg1` to see whether it contains another string `$arg2` as a substring. The result depends on whether `$arg2` is a substring of `$arg1`, and if so, on the range of characters in `$arg1` which `$arg2` matches.

When the Unicode code point collation is used, this simply involves determining whether `$arg1` contains a contiguous sequence of characters whose code points are the same, one for one, with the code points of the characters in `$arg2`.

When a collation is specified, the rules are more complex.

All collations support the capability of deciding whether two strings are considered equal, and if not, which of the strings should be regarded as preceding the other. For functions such as `fn:compare()`, this is all that is required. For other functions, such as `fn:contains()`, the collation needs to support an additional property: it must be able to decompose the string into a sequence of collation units, each unit consisting of one or more characters, such that two strings can be compared by pairwise comparison of these units. ("collation unit" is equivalent to "collation element" as defined in [Unicode Collation Algorithm].) The string `$arg1` is then considered to contain `$arg2` as a substring if the sequence of collation units corresponding to `$arg2` is a subsequence of the sequence of the collation units corresponding to `$arg1`. The characters in `$arg1` that match are the characters corresponding to these collation units.

This rule may occasionally lead to surprises. For example, consider a collation that treats "Jaeger" and "Jäger" as equal. It might do this by treating "ä" as representing two collation units, in which case the expression `fn:contains("Jäger", "eg")` will return `true`. Alternatively, a collation might treat "ae" as a single collation unit, in which case the expression `fn:contains("Jaeger", "eg")` will return `false`. The results of these functions thus depend strongly on the properties of the collation that is used. In addition, collations may specify that some collation units should be ignored during matching.

In the definitions below, we refer to the terms **match** and **minimal match** as defined in definitions DS2 and DS4 of [Unicode Collation Algorithm]. In applying these definitions:

- *C* is the collation; that is, the value of the `$collation` argument if specified, otherwise the default collation.
- *P* is the (candidate) substring `$arg2`
- *Q* is the (candidate) containing string `$arg1`
- The boundary condition *B* is satisfied at the start and end of a string, and between any two characters that belong to different collation units (collation elements in the language of [Unicode Collation Algorithm]). It is not satisfied between two characters that belong to the same collation unit.

It is possible to define collations that do not have the ability to decompose a string into units suitable for substring matching. An argument to a function defined in this section may be a URI that identifies a collation that is able to compare two strings, but that does not have the capability to split the string into collation units. Such a collation may cause the function to fail, or to give unexpected results or it may be rejected as an unsuitable argument. The ability to decompose strings into collation units is an ·implementation-defined· property of the collation.

Function	Meaning
`fn:contains`	Indicates whether one `xs:string` contains another `xs:string`. A collation may be specified.
`fn:starts-with`	Indicates whether the value of one `xs:string` begins with the collation units of another `xs:string`. A collation may be specified.
`fn:ends-with`	Indicates whether the value of one `xs:string` ends with the collation units of another `xs:string`. A collation may be specified.
`fn:substring-before`	Returns the collation units of one `xs:string` that precede in that `xs:string` the collation units of another `xs:string`. A collation may be specified.
`fn:substring-after`	Returns the collation units of `xs:string` that follow in that `xs:string` the collation units of another `xs:string`. A collation may be specified.

7.5.1 fn:contains

```
fn:contains($arg1 as xs:string?, $arg2 as xs:string?) as xs:boolean
fn:contains($arg1      as xs:string?,
            $arg2      as xs:string?,
            $collation as xs:string) as xs:boolean
```

Summary: Returns an `xs:boolean` indicating whether or not the value of `$arg1` contains (at the beginning, at the end, or anywhere within) at least one sequence of collation units that provides a minimal match to the collation units in the value of `$arg2`, according to the collation that is used.

> **Note:**
>
> "Minimal match" is defined in [Unicode Collation Algorithm].

If the value of `$arg1` or `$arg2` is the empty sequence, or contains only ignorable collation units, it is interpreted as the zero-length string.

If the value of `$arg2` is the zero-length string, then the function returns `true`.

If the value of `$arg1` is the zero-length string, the function returns `false`.

The collation used by the invocation of this function is determined according to the rules in **7.3.1 Collations**. If the specified collation does not support collation units an error ·may· be raised [err:FOCH0004].

7.5.1.1 Examples

CollationA used in these examples is a collation in which both "-" and "*" are ignorable collation units.

> **Note:**
>
> "Ignorable collation unit" is equivalent to "ignorable collation element" in [Unicode Collation Algorithm].

- `fn:contains ( "tattoo", "t")` returns `true`.
- `fn:contains ( "tattoo", "ttt")` returns `false`.
- `fn:contains ( "", ())` returns `true`. The first rule is applied, followed by the second rule.
- `fn:contains ( "abcdefghi", "-d-e-f-", "CollationA")` returns `true`.
- `fn:contains ( "a*b*c*d*e*f*g*h*i*", "d-ef-", "CollationA")` returns `true`.
- `fn:contains ( "abcd***e---f*--*ghi", "def", "CollationA")` returns `true`.
- `fn:contains ( (), "--***-*---", "CollationA")` returns `true`. The second argument contains only ignorable collation units and is equivalent to the zero-length string.

7.5.2 fn:starts-with

```
fn:starts-with($arg1 as xs:string?, $arg2 as xs:string?) as xs:boolean
fn:starts-with($arg1      as xs:string?,
               $arg2      as xs:string?,
               $collation as xs:string) as xs:boolean
```

Summary: Returns an `xs:boolean` indicating whether or not the value of `$arg1` starts with a sequence of collation units that provides a match to the collation units of `$arg2` according to the collation that is used.

Note:

"Match" is defined in [Unicode Collation Algorithm].

If the value of `$arg1` or `$arg2` is the empty sequence, or contains only ignorable collation units, it is interpreted as the zero-length string.

If the value of `$arg2` is the zero-length string, then the function returns `true`. If the value of `$arg1` is the zero-length string and the value of `$arg2` is not the zero-length string, then the function returns `false`.

The collation used by the invocation of this function is determined according to the rules in **7.3.1 Collations**. If the specified collation does not support collation units an error ·may· be raised [err:FOCH0004].

7.5.2.1 Examples

CollationA used in these examples is a collation in which both "-" and "*" are ignorable collation units.

Note:

"Ignorable collation unit" is equivalent to "ignorable collation element" in [Unicode Collation Algorithm].

- `fn:starts-with("tattoo", "tat")` returns `true`.
- `fn:starts-with ( "tattoo", "att")` returns `false`.
- `fn:starts-with ((), ())` returns `true`.
- `fn:starts-with ( "abcdefghi", "-a-b-c-", "CollationA")` returns `true`.
- `fn:starts-with ( "a*b*c*d*e*f*g*h*i*", "a-bc-", "CollationA")` returns `true`.
- `fn:starts-with ( "abcd***e---f*--*ghi", "abcdef", "CollationA")` returns `true`.
- `fn:starts-with ( (), "--***-*---", "CollationA")` returns `true`. The second argument contains only ignorable collation units and is equivalent to the zero-length string.
- `fn:starts-with ( "-abcdefghi", "-abc", "CollationA")` returns `true`.

7.5.3 fn:ends-with

```
fn:ends-with($arg1 as xs:string?, $arg2 as xs:string?) as xs:boolean
fn:ends-with($arg1      as xs:string?,
             $arg2      as xs:string?,
             $collation as xs:string) as xs:boolean
```

Summary: Returns an `xs:boolean` indicating whether or not the value of `$arg1` starts with a sequence of collation units that provides a match to the collation units of `$arg2` according to the collation that is used.

Note:

"Match" is defined in [Unicode Collation Algorithm].

If the value of `$arg1` or `$arg2` is the empty sequence, or contains only ignorable collation units, it is interpreted as the zero-length string.

If the value of `$arg2` is the zero-length string, then the function returns `true`. If the value of `$arg1` is the zero-length string and the value of `$arg2` is not the zero-length string, then the function returns `false`.

The collation used by the invocation of this function is determined according to the rules in **7.3.1 Collations**. If the specified collation does not support collation units an error ·may· be raised [err:FOCH0004].

7.5.3.1 Examples

CollationA used in these examples is a collation in which both "-" and "*" are ignorable collation units.

> **Note:**
>
> "Ignorable collation unit" is equivalent to "ignorable collation element" in [Unicode Collation Algorithm].

- `fn:ends-with ( "tattoo", "tattoo")` returns `true`.
- `fn:ends-with ( "tattoo", "atto")` returns `false`.
- `fn:ends-with ((), ())` returns `true`.
- `fn:ends-with ( "abcdefghi", "-g-h-i-", "CollationA")` returns `true`.
- `fn:ends-with ( "abcd***e---f*--*ghi", "defghi", "CollationA")` returns `true`.
- `fn:ends-with ( "abcd***e---f*--*ghi", "defghi", "CollationA")` returns `true`.
- `fn:ends-with ( (), "--***-*---", "CollationA")` returns `true`. The second argument contains only ignorable collation units and is equivalent to the zero-length string.
- `fn:ends-with ( "abcdefghi", "ghi-", "CollationA")` returns `true`.

7.5.4 fn:substring-before

```
fn:substring-before($arg1 as xs:string?, $arg2 as xs:string?) as xs:string
fn:substring-before($arg1      as xs:string?,
                    $arg2      as xs:string?,
                    $collation as xs:string) as xs:string
```

Summary: Returns the substring of the value of `$arg1` that precedes in the value of `$arg1` the first occurrence of a sequence of collation units that provides a minimal match to the collation units of `$arg2` according to the collation that is used.

> **Note:**
>
> "Minimal match" is defined in [Unicode Collation Algorithm].

If the value of `$arg1` or `$arg2` is the empty sequence, or contains only ignorable collation units, it is interpreted as the zero-length string.

If the value of `$arg2` is the zero-length string, then the function returns the zero-length string.

If the value of `$arg1` does not contain a string that is equal to the value of `$arg2`, then the function returns the zero-length string.

The collation used by the invocation of this function is determined according to the rules in **7.3.1 Collations** If the specified collation does not support collation units an error ·may· be raised [err:FOCH0004].

7.5.4.1 Examples

CollationA used in these examples is a collation in which both "-" and "*" are ignorable collation units.

> **Note:**
>
> "Ignorable collation unit" is equivalent to "ignorable collation element" in [Unicode Collation Algorithm].

- `fn:substring-before ( "tattoo", "attoo")` returns "`t`".
- `fn:substring-before ( "tattoo", "tatto")` returns "".
- `fn:substring-before ((), ())` returns "".
- `fn:substring-before ( "abcdefghi", "--d-e-", "CollationA")` returns "`abc`".
- `fn:substring-before ( "abc--d-e-fghi", "--d-e-", "CollationA")` returns "`abc--`".
- `fn:substring-before ( "a*b*c*d*e*f*g*h*i*", "***cde", "CollationA")` returns "`a*b*`".
- `fn:substring-before ( "Eureka!", "--***-*---", "CollationA")` returns "". The second argument contains only ignorable collation units and is equivalent to the zero-length string.

7.5.5 fn:substring-after

```
fn:substring-after($arg1 as xs:string?, $arg2 as xs:string?) as xs:string
fn:substring-after($arg1      as xs:string?,
                   $arg2      as xs:string?,
                   $collation as xs:string) as xs:string
```

Summary: Returns the substring of the value of `$arg1` that follows in the value of `$arg1` the first occurrence of a sequence of collation units that provides a minimal match to the collation units of `$arg2` according to the collation that is used.

> **Note:**
>
> "Minimal match" is defined in [Unicode Collation Algorithm].

If the value of `$arg1` or `$arg2` is the empty sequence, or contains only ignorable collation units, it is interpreted as the zero-length string.

If the value of `$arg2` is the zero-length string, then the function returns the value of `$arg1`.

If the value of `$arg1` does not contain a string that is equal to the value of `$arg2`, then the function returns the zero-length string.

The collation used by the invocation of this function is determined according to the rules in **7.3.1 Collations** If the specified collation does not support collation units an error ·may· be raised [err:FOCH0004].

7.5.5.1 Examples

CollationA used in these examples is a collation in which both "-" and "*" are ignorable collation units.

> **Note:**

"Ignorable collation unit" is equivalent to "ignorable collation element" in [Unicode Collation Algorithm].

- `fn:substring-after("tattoo", "tat")` returns "`too`".
- `fn:substring-after ( "tattoo", "tattoo")` returns "".
- `fn:substring-after ((), ())` returns "".
- `fn:substring-after ( "abcdefghi", "--d-e-", "CollationA")` returns "`fghi`".
- `fn:substring-after ( "abc--d-e-fghi", "--d-e-", "CollationA")` returns "`-fghi` ".
- `fn:substring-after ( "a*b*c*d*e*f*g*h*i*", "***cde***", "CollationA")` returns "`*f*g*h*i*`".
- `fn:substring-after ( "Eureka!", "--***-*---", "CollationA")` returns "`Eureka!`". The second argument contains only ignorable collation units and is equivalent to the zero-length string.

7.6 String Functions that Use Pattern Matching

The three functions described in this section make use of a regular expression syntax for pattern matching. This is described below.

Function	Meaning
`fn:matches`	Returns an `xs:boolean` value that indicates whether the value of the first argument is matched by the regular expression that is the value of the second argument.
`fn:replace`	Returns the value of the first argument with every substring matched by the regular expression that is the value of the second argument replaced by the replacement string that is the value of the third argument.
`fn:tokenize`	Returns a sequence of one or more `xs:string`s whose values are substrings of the value of the first argument separated by substrings that match the regular expression that is the value of the second argument.

7.6.1 Regular Expression Syntax

The regular expression syntax used by these functions is defined in terms of the regular expression syntax specified in XML Schema (see [XML Schema Part 2: Datatypes Second Edition]), which in turn is based on the established conventions of languages such as Perl. However, because XML Schema uses regular expressions only for validity checking, it omits some facilities that are widely-used with languages such as Perl. This section, therefore, describes extensions to the XML Schema regular expressions syntax that reinstate these capabilities.

Note:

It is recommended that implementers consult [Unicode Regular Expressions] for information on using regular expression processing on Unicode characters.

The regular expression syntax and semantics are identical to those defined in [XML Schema Part 2: Datatypes Second Edition] with the following additions:

- Two meta-characters, `^` and `$` are added. By default, the meta-character `^` matches the start of the entire string, while `$` matches the end of the entire string. In multi-line mode, `^` matches the start of any line (that is, the start of the entire string, and the position immediately after a newline character), while `$` matches the end of any line (that is, the end of the entire string, and the position immediately before a newline character). Newline here means the character `#x0A` only.
  This means that the production in [XML Schema Part 2: Datatypes Second Edition]:
  ```
  [10] Char ::= [^.\?*+()|#x5B#x5D]
  ```
  is modified to read:

```
[10] Char ::= [^.\?*+{}()|^$#x5B#x5D]
```

The characters `#x5B` and `#x5D` correspond to "`[`" and "`]`" respectively.

> **Note:**
>
> The definition of Char (production [10]) in [XML Schema Part 2: Datatypes Second Edition] has a known error in which it omits the left brace ("{") and right brace ("}"). That error is corrected here.

The following production:

```
[11] charClass ::= charClassEsc | charClassExpr | WildCardEsc
```

is modified to read:

```
[11] charClass ::= charClassEsc | charClassExpr | WildCardEsc | "^" | "$"
```

- *Reluctant quantifiers* are supported. They are indicated by a " `?` " following a quantifier. Specifically:
  - `X??` matches X, once or not at all
  - `X*?` matches X, zero or more times
  - `X+?` matches X, one or more times
  - `X{n}?` matches X, exactly n times
  - `X{n,}?` matches X, at least n times
  - `X{n,m}?` matches X, at least n times, but not more than m times

  The effect of these quantifiers is that the regular expression matches the *shortest* possible substring consistent with the match as a whole succeeding. Without the " `?` ", the regular expression matches the *longest* possible substring.

  To achieve this, the production in [XML Schema Part 2: Datatypes Second Edition]:

  ```
  [4] quantifier ::= [?*+] | ( '{' quantity '}' )
  ```

  is changed to:

  ```
  [4] quantifier ::= ( [?*+] | ( '{' quantity '}' ) ) '?'?
  ```

  > **Note:**
  >
  > Reluctant quantifiers have no effect on the results of the boolean `fn:matches` function, since this function is only interested in discovering whether a match exists, and not where it exists.

- Sub-expressions (groups) within the regular expression are recognized. The regular expression syntax defined by [XML Schema Part 2: Datatypes Second Edition] allows a regular expression to contain parenthesized sub-expressions, but attaches no special significance to them. The `fn:replace()` function described below allows access to the parts of the input string that matched a sub-expression (called captured substrings). The sub-expressions are numbered according to the position of the opening parenthesis in left-to-right order within the top-level regular expression: the first opening parenthesis identifies captured substring 1, the second identifies captured substring 2, and so on. 0 identifies the substring captured by the entire regular expression. If a sub-expression matches more than one substring (because it is within a construct that allows repetition), then only the *last* substring that it matched will be captured.

- Back-references are allowed outside a character class expression. A back-reference is an additional kind of atom. The construct `\N` where `N` is a single digit is always recognized as a back-reference; if this is followed by further digits, these digits are taken to be part of the back-reference if and only if the resulting number `NN` is such that the back-reference is preceded by `NN` or more unescaped opening parentheses. The regular expression is invalid if a back-reference refers to a subexpression that does not exist or whose closing right parenthesis occurs after the back-reference.

  A back-reference matches the string that was matched by the `N`th capturing subexpression within the regular expression, that is, the parenthesized subexpression whose opening left parenthesis is the `N`th unescaped left parenthesis within the regular expression. For example, the regular expression `('|").*\1` matches a sequence of characters delimited either by an apostrophe at the start and end, or by a quotation mark at the start and end.

If no string is matched by the `N`th capturing subexpression, the back-reference is interpreted as matching a zero-length string.

Back-references change the following production:

```
[9] atom ::= Char | charClass | ( '(' regExp ')' )
```

to

```
[9] atom ::= Char | charClass | ( '(' regExp ')' ) | backReference
```

```
[9a] backReference ::= "\" [1-9][0-9]*
```

Note:

Within a character class expression, `\` followed by a digit is invalid. Some other regular expression languages interpret this as an octal character reference.

- Single character escapes are extended to allow the `$` character to be escaped. The following production is changed:

```
[24]SingleCharEsc ::= '\' [nrt\|.?*+(){}#x2D#x5B#x5D#x5E]
```

to

```
[24]SingleCharEsc ::= '\' [nrt\|.?*+(){}$#x2D#x5B#x5D#x5E]
```

7.6.1.1 Flags

All these functions provide an optional parameter, `$flags`, to set options for the interpretation of the regular expression. The parameter accepts a `xs:string`, in which individual letters are used to set options. The presence of a letter within the string indicates that the option is on; its absence indicates that the option is off. Letters may appear in any order and may be repeated. If there are characters present that are not defined here as flags, then an error is raised [err:FORX0001].

The following options are defined:

- `s`: If present, the match operates in "dot-all" mode. (Perl calls this the single-line mode.) If the `s` flag is not specified, the meta-character `.` matches any character except a newline (`#x0A`) character. In dot-all mode, the meta-character `.` matches any character whatsoever. Suppose the input contains "hello" and "world" on two lines. This will not be matched by the regular expression "hello.*world" unless dot-all mode is enabled.
- `m`: If present, the match operates in multi-line mode. By default, the meta-character `^` matches the start of the entire string, while $ matches the end of the entire string. In multi-line mode, `^` matches the start of any line (that is, the start of the entire string, and the position immediately after a newline character other than a newline that appears as the last character in the string), while `$` matches the end of any line (that is, the position immediately before a newline character, and the end of the entire string if there is no newline character at the end of the string). Newline here means the character `#x0A` only.
- `i`: If present, the match operates in case-insensitive mode. The detailed rules are as follows. In these rules, a character C2 is considered to be a *case-variant* of another character C1 if the following XPath expression returns `true` when the two characters are considered as strings of length one, and the Unicode codepoint collation is used:

  `fn:lower-case(C1) eq fn:lower-case(C2)`

  or

  `fn:upper-case(C1) eq fn:upper-case(C2)`

  Note that the case-variants of a character under this definition are always single characters.

  1. When a normal character (`Char`) is used as an atom, it represents the set containing that character and all its case-variants. For example, the regular expression "z" will match both "z" and "Z".

2. A character range (`charRange`) represents the set containing all the characters that it would match in the absence of the "`i`" flag, together with their case-variants. For example, the regular expression "[A-Z]" will match all the letters A-Z and all the letters a-z. It will also match certain other characters such as `#x212A` (KELVIN SIGN), since `fn:lower-case("#x212A")` is "k".

   This rule applies also to a character range used in a character class subtraction (`charClassSub`): thus [A-Z-[IO]] will match characters such as "A", "B", "a", and "b", but will not match "I", "O", "i", or "o".

   The rule also applies to a character range used as part of a negative character group: thus [^Q] will match every character except "Q" and "q" (these being the only case-variants of "Q" in Unicode).
3. A back-reference is compared using case-blind comparison: that is, each character must either be the same as the corresponding character of the previously matched string, or must be a case-variant of that character. For example, the strings "Mum", "mom", "Dad", and "DUD" all match the regular expression "([md])[aeiou]\1" when the "`i`" flag is used.
4. All other constructs are unaffected by the "`i`" flag. For example, "\p{Lu}" continues to match upper-case letters only.

- `x`: If present, whitespace characters (#x9, #xA, #xD and #x20) in the regular expression are removed prior to matching with one exception: whitespace characters within character class expressions (`charClassExpr`) are not removed. This flag can be used, for example, to break up long regular expressions into readable lines.

  Examples:

  `fn:matches("helloworld", "hello world", "x")` returns `true`

  `fn:matches("helloworld", "hello[ ]world", "x")` returns `false`

  `fn:matches("hello world", "hello\ sworld", "x")` returns `true`

  `fn:matches("hello world", "hello world", "x")` returns `false`

7.6.2 fn:matches

```
fn:matches($input as xs:string?, $pattern as xs:string) as xs:boolean
fn:matches($input   as xs:string?,
           $pattern as xs:string,
           $flags   as xs:string) as xs:boolean
```

Summary: The function returns `true` if `$input` matches the regular expression supplied as `$pattern` as influenced by the value of `$flags`, if present; otherwise, it returns `false`.

The effect of calling the first version of this function (omitting the argument `$flags`) is the same as the effect of calling the second version with the `$flags` argument set to a zero-length string. Flags are defined in **7.6.1.1 Flags**.

If `$input` is the empty sequence, it is interpreted as the zero-length string.

Unless the metacharacters `^` and `$` are used as anchors, the string is considered to match the pattern if any substring matches the pattern. But if anchors are used, the anchors must match the start/end of the string (in string mode), or the start/end of a line (in multiline mode).

> **Note:**
>
> This is different from the behavior of patterns in [XML Schema Part 2: Datatypes Second Edition], where regular expressions are *implicitly* anchored.

An error is raised [err:FORX0002] if the value of `$pattern` is invalid according to the rules described in section **7.6.1 Regular Expression Syntax**.

An error is raised [err:FORX0001] if the value of `$flags` is invalid according to the rules described in section **7.6.1 Regular Expression Syntax**.

7.6.2.1 Examples

- `fn:matches("abracadabra", "bra")` returns true
- `fn:matches("abracadabra", "^a.*a$")` returns true
- `fn:matches("abracadabra", "^bra")` returns false

Given the source document:

```
<poem author="Wilhelm Busch">
Kaum hat dies der Hahn gesehen,
Fängt er auch schon an zu krähen:
«Kikeriki! Kikikerikih!!»
Tak, tak, tak! - da kommen sie.
</poem>
```

the following function calls produce the following results, with the `poem` element as the context node:

- `fn:matches(., "Kaum.*krähen")` returns `false`
- `fn:matches(., "Kaum.*krähen", "s")` returns `true`
- `fn:matches(., "^Kaum.*gesehen,$", "m")` returns `true`
- `fn:matches(., "^Kaum.*gesehen,$")` returns `false`
- `fn:matches(., "kiki", "i")` returns `true`

Note:

Regular expression matching is defined on the basis of Unicode code points; it takes no account of collations.

7.6.3 fn:replace

```
fn:replace($input       as xs:string?,
           $pattern     as xs:string,
           $replacement as xs:string) as xs:string
fn:replace($input       as xs:string?,
           $pattern     as xs:string,
           $replacement as xs:string,
           $flags       as xs:string) as xs:string
```

Summary: The function returns the `xs:string` that is obtained by replacing each non-overlapping substring of `$input` that matches the given `$pattern` with an occurrence of the `$replacement` string.

The effect of calling the first version of this function (omitting the argument `$flags`) is the same as the effect of calling the second version with the `$flags` argument set to a zero-length string. Flags are defined in **7.6.1.1 Flags**.

The `$flags` argument is interpreted in the same manner as for the `fn:matches()` function.

If `$input` is the empty sequence, it is interpreted as the zero-length string.

If two overlapping substrings of `$input` both match the `$pattern`, then only the first one (that is, the one whose first character comes first in the `$input` string) is replaced.

Within the `$replacement` string, a variable `$N` may be used to refer to the substring captured by the Nth parenthesized sub-expression in the regular expression. For each match of the pattern, these variables are assigned the value of the content matched by the relevant sub-expression, and the modified replacement string is then substituted for the characters in `$input` that matched the pattern. `$0` refers to the substring captured by the regular expression as a whole.

More specifically, the rules are as follows, where `S` is the number of parenthesized sub-expressions in the regular expression, and `N` is the decimal number formed by taking all the digits that consecutively follow the `$` character:

1. If `N=0`, then the variable is replaced by the substring matched by the regular expression as a whole.
2. If `1<=N<=S`, then the variable is replaced by the substring captured by the Nth parenthesized sub-expression. If the `Nth` parenthesized sub-expression was not matched, then the variable is replaced by the zero-length string.
3. If `S<N<=9`, then the variable is replaced by the zero-length string.
4. Otherwise (if `N>S` and `N>9`), the last digit of `N` is taken to be a literal character to be included "as is" in the replacement string, and the rules are reapplied using the number `N` formed by stripping off this last digit.

For example, if the replacement string is "`$23`" and there are 5 substrings, the result contains the value of the substring that matches the second sub-expression, followed by the digit "`3`".

A literal "`$`" symbol must be written as "`\$`".

A literal "`\`" symbol must be written as "`\\`".

If two alternatives within the pattern both match at the same position in the `$input`, then the match that is chosen is the one matched by the first alternative. For example:

```
fn:replace("abcd", "(ab)|(a)", "[1=$1][2=$2]") returns "[1=ab][2=]cd"
```

An error is raised [err:FORX0002] if the value of `$pattern` is invalid according to the rules described in section **7.6.1 Regular Expression Syntax**.

An error is raised [err:FORX0001] if the value of `$flags` is invalid according to the rules described in section **7.6.1 Regular Expression Syntax**.

An error is raised [err:FORX0003] if the pattern matches a zero-length string, that is, if the expression `fn:matches("", $pattern, $flags)` returns `true`. It is not an error, however, if a captured substring is zero-length.

An error is raised [err:FORX0004] if the value of `$replacement` contains a "`$`" character that is not immediately followed by a digit `0-9` and not immediately preceded by a "\".

An error is raised [err:FORX0004] if the value of `$replacement` contains a "`\`" character that is not part of a "`\\`" pair, unless it is immediately followed by a "`$`" character.

7.6.3.1 Examples

- `replace("abracadabra", "bra", "*")` returns `"a*cada*"`
- `replace("abracadabra", "a.*a", "*")` returns `"*"`

- `replace("abracadabra", "a.*?a", "*")` returns `"*c*bra"`
- `replace("abracadabra", "a", "")` returns `"brcdbr"`
- `replace("abracadabra", "a(.)", "a$1$1")` returns `"abbraccaddabbra"`
- `replace("abracadabra", ".*?", "$1")` raises an error, because the pattern matches the zero-length string
- `replace("AAAA", "A+", "b")` returns `"b"`
- `replace("AAAA", "A+?", "b")` returns `"bbbb"`
- `replace("darted", "^(.*?)d(.*)$", "$1c$2")` returns `"carted"`. The first `d` is replaced.

7.6.4 fn:tokenize

```
fn:tokenize($input as xs:string?, $pattern as xs:string) as xs:string*
fn:tokenize($input   as xs:string?,
            $pattern as xs:string,
            $flags   as xs:string) as xs:string*
```

Summary: This function breaks the `$input` string into a sequence of strings, treating any substring that matches `$pattern` as a separator. The separators themselves are not returned.

The effect of calling the first version of this function (omitting the argument `$flags`) is the same as the effect of calling the second version with the `$flags` argument set to a zero-length string. Flags are defined in **7.6.1.1 Flags**.

The `$flags` argument is interpreted in the same way as for the `fn:matches()` function.

If `$input` is the empty sequence, or if `$input` is the zero-length string, the result is the empty sequence.

If the supplied `$pattern` matches a zero-length string, that is, if `fn:matches("", $pattern, $flags)` returns `true`, then an error is raised: [err:FORX0003].

If a separator occurs at the start of the `$input` string, the result sequence will start with a zero-length string. Zero-length strings will also occur in the result sequence if a separator occurs at the end of the `$input` string, or if two adjacent substrings match the supplied `$pattern`.

If two alternatives within the supplied `$pattern` both match at the same position in the `$input` string, then the match that is chosen is the first. For example:

```
fn:tokenize("abracadabra", "(ab)|(a)") returns ("", "r", "c", "d", "r", "")
```

An error is raised [err:FORX0002] if the value of `$pattern` is invalid according to the rules described in section **7.6.1 Regular Expression Syntax**.

An error is raised [err:FORX0001] if the value of `$flags` is invalid according to the rules described in section **7.6.1 Regular Expression Syntax**.

7.6.4.1 Examples

- `fn:tokenize("The cat sat on the mat", "\s+")` returns `("The", "cat", "sat", "on", "the", "mat")`
- `fn:tokenize("1, 15, 24, 50", ",\s*")` returns `("1", "15", "24", "50")`
- `fn:tokenize("1,15,,24,50,", ",")` returns `("1", "15", "", "24", "50", "")`
- `fn:tokenize("abba", ".?")` raises the error [err:FORX0003].

- `fn:tokenize("Some unparsed <br> HTML <BR> text", "\s*<br>\s*", "i")` returns `("Some unparsed", "HTML", "text")`

8 Functions on anyURI

This section specifies functions that take anyURI as arguments.

Function	Meaning
`fn:resolve-uri`	Returns an `xs:anyURI` representing an absolute `xs:anyURI` given a base URI and a relative URI.

8.1 fn:resolve-uri

```
fn:resolve-uri($relative as xs:string?) as xs:anyURI?
```

```
fn:resolve-uri($relative as xs:string?, $base as xs:string) as xs:anyURI?
```

Summary: This function enables a relative URI reference to be resolved against an absolute URI.

The first form of this function resolves `$relative` against the value of the base-uri property from the static context. If the base-uri property is not initialized in the static context an error is raised [err:FONS0005].

If `$relative` is a relative URI reference, it is resolved against `$base`, or against the base-uri property from the static context, using an algorithm such as those described in [RFC 2396] or [RFC 3986], and the resulting absolute URI reference is returned.

If `$relative` is an absolute URI reference, it is returned unchanged.

If `$relative` is the empty sequence, the empty sequence is returned.

If `$relative` is not a valid URI according to the rules of the `xs:anyURI` data type, or if it is not a suitable relative reference to use as input to the chosen resolution algorithm, then an error is raised [err:FORG0002].

If `$base` is not a valid URI according to the rules of the `xs:anyURI` data type, if it is not a suitable URI to use as input to the chosen resolution algorithm (for example, if it is a relative URI reference, if it is a non-hierarchic URI, or if it contains a fragment identifier), then an error is raised [err:FORG0002].

If the chosen resolution algorithm fails for any other reason then an error is raised [err:FORG0009].

> **Note:**
>
> Resolving a URI does not dereference it. This is merely a syntactic operation on two character strings.
>
> **Note:**
>
> The algorithms in the cited RFCs include some variations that are optional or recommended rather than mandatory; they also describe some common practices that are not recommended, but which are permitted for backwards compatibility. Where the cited RFCs permit variations in behavior, so does this specification.

9 Functions and Operators on Boolean Values

This section defines functions and operators on the [XML Schema Part 2: Datatypes Second Edition] boolean datatype.

9.1 Additional Boolean Constructor Functions

The following additional constructor functions are defined on the boolean type.

Function	Meaning
`fn:true`	Constructs the xs:boolean value 'true'.
`fn:false`	Constructs the xs:boolean value 'false'.

9.1.1 fn:true

```
fn:true() as xs:boolean
```

Summary: Returns the `xs:boolean` value `true`. Equivalent to `xs:boolean("1")`.

9.1.1.1 Examples

- `fn:true()` returns `true`.

9.1.2 fn:false

```
fn:false() as xs:boolean
```

Summary: Returns the `xs:boolean` value `false`. Equivalent to `xs:boolean("0")`.

9.1.2.1 Examples

- `fn:false()` returns `false`.

9.2 Operators on Boolean Values

The following functions define the semantics of operators on boolean values in [XQuery 1.0: An XML Query Language] and [XML Path Language (XPath) 2.0]:

Operator	Meaning
`op:boolean-equal`	Equality of `xs:boolean` values
`op:boolean-less-than`	A less-than operator on `xs:boolean` values: `false` is less than `true`.
`op:boolean-greater-than`	A greater-than operator on `xs:boolean` values: `true` is greater than `false`.

The ordering operators `op:boolean-less-than` and `op:boolean-greater-than` are provided for application purposes and for compatibility with [XML Path Language (XPath) Version 1.0]. The [XML Schema Part 2: Datatypes Second Edition] datatype `xs:boolean` is not ordered.

9.2.1 op:boolean-equal

```
op:boolean-equal($value1 as xs:boolean, $value2 as xs:boolean) as xs:boolean
```

Summary: Returns `true` if both arguments are `true` or if both arguments are `false`. Returns `false` if one of the arguments is `true` and the other argument is `false`.

This function backs up the "eq" operator on `xs:boolean` values.

9.2.2 op:boolean-less-than

```
op:boolean-less-than($arg1 as xs:boolean, $arg2 as xs:boolean) as xs:boolean
```

Summary: Returns `true` if `$arg1` is `false` and `$arg2` is `true`. Otherwise, returns `false`.

This function backs up the "lt" and "ge" operators on `xs:boolean` values.

9.2.3 op:boolean-greater-than

```
op:boolean-greater-than($arg1 as xs:boolean, $arg2 as xs:boolean) as xs:boolean
```

Summary: Returns `true` if `$arg1` is `true` and `$arg2` is `false`. Otherwise, returns `false`.

This function backs up the "gt" and "le" operators on `xs:boolean` values.

9.3 Functions on Boolean Values

The following functions are defined on boolean values:

Function	Meaning
`fn:not`	Inverts the `xs:boolean` value of the argument.

9.3.1 fn:not

```
fn:not($arg as item()*) as xs:boolean
```

Summary: `$arg` is first reduced to an effective boolean value by applying the `fn:boolean()` function. Returns `true` if the effective boolean value is `false`, and `false` if the effective boolean value is `true`.

9.3.1.1 Examples

- `fn:not(fn:true())` returns `false`.
- `fn:not("false")` returns `false`.

10 Functions and Operators on Durations, Dates and Times

This section discusses operations on the [XML Schema Part 2: Datatypes Second Edition] date and time types. It also discusses operations on two subtypes of `xs:duration` that are defined in Section 2.6 Types*DM*. See **10.3 Two Totally Ordered Subtypes of Duration**.

See [Working With Timezones] for a disquisition on working with date and time values with and without timezones.

10.1 Duration, Date and Time Types

The operators described in this section are defined on the following date and time types:

- xs:dateTime
- xs:date
- xs:time
- xs:gYearMonth
- xs:gYear
- xs:gMonthDay
- xs:gMonth
- xs:gDay

Note that only equality is defined on `xs:gYearMonth`, `xs:gYear`, `xs:gMonthDay`, `xs:gMonth` and `xs:gDay` values.

In addition, operators are defined on:

- xs:duration

and on the **10.3 Two Totally Ordered Subtypes of Duration**:

- xs:yearMonthDuration
- xs:dayTimeDuration

Note that no ordering relation is defined on `xs:duration` values.Two `xs:duration` values may however be compared for equality. Operations on durations (including equality comparison, casting to string, and extraction of components) all treat the duration as normalized. This means that the seconds and minutes components will always be less than 60, the hours component less than 24, and the months component less than 12. Thus, for example, a duration of 120 seconds always gives the same result as a duration of two minutes.

10.1.1 Limits and Precision

For a number of the above datatypes [XML Schema Part 2: Datatypes Second Edition] extends the basic [ISO 8601] lexical representations, such as YYYY-MM-DDThh:mm:ss.s for dateTime, by allowing a preceding minus sign, more than four digits to represent the year field — no maximum is specified — and an unlimited number of digits for fractional seconds. Leap seconds are not supported.

All *minimally conforming* processors ·must· support positive year values with a minimum of 4 digits (i.e., YYYY) and a minimum fractional second precision of 1 millisecond or three digits (i.e., s.sss). However, *conforming* processors ·may· set larger ·implementation-defined· limits on the maximum number of digits they support in these two situations. Processors ·may· also choose to support the year 0000 and years with negative values. The results of operations on dates that cross the year 0000 are ·implementation-defined·.

A processor that limits the number of digits in date and time datatype representations may encounter overflow and underflow conditions when it tries to execute the functions in **10.8 Arithmetic Operators on Durations, Dates and Times**. In these situations, the processor ·must· return P0M or PT0S in case of duration underflow and 00:00:00 in case of time underflow. It ·must· raise an error [err:FODT0001] in case of overflow.

The value spaces of the two totally ordered subtypes of `xs:duration` described in **10.3 Two Totally Ordered Subtypes of Duration** are `xs:integer` months for `xs:yearMonthDuration` and `xs:decimal` seconds for `xs:dayTimeDuration`. If a processor limits the number of digits allowed in the representation of `xs:integer` and `xs:decimal` then overflow and underflow situations can arise when it tries to execute the functions in **10.6 Arithmetic Operators on Durations**. In these situations the processor ·must· return zero in case of numeric underflow and P0M or PT0S in case of duration underflow. It ·must· raise an error [err:FODT0002] in case of overflow.

10.2 Date/time datatype values

As defined in Section 3.3.2 Dates and Times^DM, `xs:dateTime`, `xs:date`, `xs:time`, `xs:gYearMonth`, `xs:gYear`, `xs:gMonthDay`, `xs:gMonth`, `xs:gDay` values, referred to collectively as date/time values, are represented as seven components or properties: `year`, `month`, `day`, `hour`, `minute`, `second` and `timezone`. The value of the first five components are `xs:integer`s. The value of the `second` component is an `xs:decimal` and the value of the `timezone` component is an `xs:dayTimeDuration`. For all the date/time datatypes, the `timezone` property is optional and may or may not be present. Depending on the datatype, some of the remaining six properties must be present and some must be absent. Absent, or missing, properties are represented by the empty sequence. This value is referred to as the *local* value in that the value is in the given timezone. Before comparing or subtracting `xs:dateTime` values, this local value ·must· be translated or *normalized* to UTC.

For `xs:time`, `"00:00:00"` and `"24:00:00"` are alternate lexical forms for the same value, whose canonical representation is `"00:00:00"`. For `xs:dateTime`, a time component `"24:00:00"` translates to `"00:00:00"` of the following day.

10.2.1 Examples

- An `xs:dateTime` with lexical representation `1999-05-31T05:00:00` is represented in the datamodel by `{1999, 5, 31, 5, 0, 0.0, ()}`.
- An `xs:dateTime` with lexical representation `1999-05-31T13:20:00-05:00` is represented by `{1999, 5, 31, 13, 20, 0.0, -PT5H}`.
- An `xs:dateTime` with lexical representation `1999-12-31T24:00:00` is represented by `{2000, 1, 1, 0, 0, 0.0, ()}`.
- An `xs:date` with lexical representation `2005-02-28+8:00` is represented by `{2005, 2, 28, (), (), (), PT8H}`.
- An `xs:time` with lexical representation `24:00:00` is represented by `{(), (), (), 0, 0, 0, ()}`.

10.3 Two Totally Ordered Subtypes of Duration

Two totally ordered subtypes of `xs:duration` are defined in Section 2.6 Types^DM specification using the mechanisms described in [XML Schema Part 2: Datatypes Second Edition] for defining user-defined types. Additional details about these types is given below.

10.3.1 xs:yearMonthDuration

[Definition] `xs:yearMonthDuration` is derived from `xs:duration` by restricting its lexical representation to contain only the year and month components. The value space of `xs:yearMonthDuration` is the set of `xs:integer` month values. The year and month components of `xs:yearMonthDuration` correspond to the Gregorian year and month components defined in section 5.5.3.2 of [ISO 8601], respectively.

10.3.1.1 Lexical representation

The lexical representation for `xs:yearMonthDuration` is the [ISO 8601] reduced format PnYnM, where nY represents the number of years and nM the number of months. The values of the years and months components are not restricted but allow an arbitrary unsigned `xs:integer`.

An optional preceding minus sign ('-') is allowed to indicate a negative duration. If the sign is omitted a positive duration is indicated. To indicate a `xs:yearMonthDuration` of 1 year, 2 months, one would write: P1Y2M. One could also indicate a `xs:yearMonthDuration` of minus 13 months as: -P13M.

Reduced precision and truncated representations of this format are allowed provided they conform to the following:

If the number of years or months in any expression equals zero (0), the number and its corresponding designator [·may·] be omitted. However, at least one number and its designator [·must·] be present. For example, P1347Y and P1347M are allowed; P-1347M is not allowed, although -P1347M is allowed. P1Y2MT is not allowed. Also, P24YM is not allowed, nor is PY43M since Y must have at least one preceding digit and M must have one preceding digit.

10.3.1.2 Calculating the value from the lexical representation

The value of a `xs:yearMonthDuration` lexical form is obtained by multiplying the value of the years component by 12 and adding the value of the months component. The value is positive or negative depending on the preceding sign.

10.3.1.3 Canonical representation

The canonical representation of `xs:yearMonthDuration` restricts the value of the months component to `xs:integer` values between 0 and 11, both inclusive. To convert from a non-canonical representation to the canonical representation, the lexical representation is first converted to a value in `xs:integer` months as defined above. This value is then divided by 12 to obtain the value of the years component of the canonical representation. The remaining number of months is the value of the months component of the canonical representation. For negative durations, the canonical form is calculated using the absolute value of the duration and a negative sign is prepended to it. If a component has the value zero (0), then the number and the designator for that component [·must·] be omitted. However, if the value is zero (0) months, the canonical form is "P0M".

10.3.1.4 Order relation on xs:yearMonthDuration

Let the function that calculates the value of an `xs:yearMonthDuration` in the manner described above be called V(d). Then for two `xs:yearMonthDuration` values x and y, x > y if and only if V(x) > V(y). The order relation on `yearMonthDuration` is a total order.

10.3.2 xs:dayTimeDuration

[Definition] `xs:dayTimeDuration` is derived from `xs:duration` by restricting its lexical representation to contain only the days, hours, minutes and seconds components. The value space of `xs:dayTimeDuration` is the set of fractional second values. The components of `xs:dayTimeDuration` correspond to the day, hour, minute and second components defined in Section 5.5.3.2 of [ISO 8601], respectively.

10.3.2.1 Lexical representation

The lexical representation for `xs:dayTimeDuration` is the [ISO 8601] truncated format PnDTnHnMnS, where nD represents the number of days, T is the date/time separator, nH the number of hours, nM the number of minutes and nS the number of seconds.

The values of the days, hours and minutes components are not restricted, but allow an arbitrary unsigned `xs:integer`. Similarly, the value of the seconds component allows an arbitrary unsigned `xs:decimal`. An optional minus sign ('-') is allowed to precede the 'P', indicating a negative duration. If the sign is omitted, the duration is positive. See also [ISO 8601] Date and Time Formats.

For example, to indicate a duration of 3 days, 10 hours and 30 minutes, one would write: P3DT10H30M. One could also indicate a duration of minus 120 days as: -P120D. Reduced precision and truncated representations of this format are allowed, provided they conform to the following:

- If the number of days, hours, minutes, or seconds in any expression equals zero (0), the number and its corresponding designator ·may· be omitted. However, at least one number and its designator ·must· be present.
- The seconds part ·may· have a decimal fraction.
- The designator 'T' ·must· be absent if and only if all of the time items are absent. The designator 'P' ·must· always be present.

For example, P13D, PT47H, P3DT2H, -PT35.89S and P4DT251M are all allowed. P-134D is not allowed (invalid location of minus sign), although -P134D is allowed.

10.3.2.2 Calculating the value of a xs:dayTimeDuration from the lexical representation

The value of a `xs:dayTimeDuration` lexical form in fractional seconds is obtained by converting the days, hours, minutes and seconds value to fractional seconds using the conversion rules: 24 hours = 1 day, 60 minutes = 1 hour and 60 seconds = 1 minute.

10.3.2.3 Canonical representation

The canonical representation of `xs:dayTimeDuration` restricts the value of the hours component to `xs:integer` values between 0 and 23, both inclusive; the value of the minutes component to `xs:integer` values between 0 and 59; both inclusive; and the value of the seconds component to `xs:decimal` valued from 0.0 to 59.999... (see [XML Schema Part 2: Datatypes Second Edition], Appendix D).

To convert from a non-canonical representation to the canonical representation, the value of the lexical form in fractional seconds is first calculated in the manner described above. The value of the days component in the canonical form is then calculated by dividing the value by 86,400 (24*60*60). The remainder is in fractional seconds. The value of the hours component in the canonical form is calculated by dividing this remainder by 3,600 (60*60). The remainder is again in fractional seconds. The value of the minutes component in the canonical form is calculated by dividing this remainder by 60. The remainder in fractional seconds is the value of the seconds component in the canonical form. For negative durations, the canonical form is calculated using the absolute value of the duration and a negative sign is prepended to it. If a component has the value zero (0) then the number and the designator for that component must be omitted. However, if all the components of the lexical form are zero (0), the canonical form is "PT0S".

10.3.2.4 Order relation on xs:dayTimeDuration

Let the function that calculates the value of a `xs:dayTimeDuration` in the manner described above be called *V(d)*. Then for two `xs:dayTimeDuration` values *x* and *y, x* > *y* if and only if *V(x)* > *V(y)*. The order relation on `xs:dayTimeDuration` is a total order.

10.4 Comparison Operators on Duration, Date and Time Values

Operator	Meaning
op:yearMonthDuration-less-than	Less-than comparison on xs:yearMonthDuration values
op:yearMonthDuration-greater-than	Greater-than comparison on xs:yearMonthDuration values
op:dayTimeDuration-less-than	Less-than comparison on xs:dayTimeDuration values
op:dayTimeDuration-greater-than	Greater-than comparison on xs:dayTimeDuration values
op:duration-equal	Equality comparison on xs:duration values
op:dateTime-equal	Equality comparison on xs:dateTime values
op:dateTime-less-than	Less-than comparison on xs:dateTime values
op:dateTime-greater-than	Greater-than comparison on xs:dateTime values
op:date-equal	Equality comparison on xs:date values
op:date-less-than	Less-than comparison on xs:date values
op:date-greater-than	Greater-than comparison on xs:date values
op:time-equal	Equality comparison on xs:time values
op:time-less-than	Less-than comparison on xs:time values
op:time-greater-than	Greater-than comparison on xs:time values
op:gYearMonth-equal	Equality comparison on xs:gYearMonth values
op:gYear-equal	Equality comparison on xs:gYear values
op:gMonthDay-equal	Equality comparison on xs:gMonthDay values
op:gMonth-equal	Equality comparison on xs:gMonth values
op:gDay-equal	Equality comparison on xs:gDay values

The following comparison operators are defined on the [XML Schema Part 2: Datatypes Second Edition] date, time and duration datatypes. Each operator takes two operands of the same type and returns an xs:boolean result. As discussed in [XML Schema Part 2: Datatypes Second Edition], the order relation on xs:duration is not a total order but, rather, a partial order. For this reason, only equality is defined on xs:duration. A full complement of comparison and arithmetic functions are defined on the two subtypes of duration described in **10.3 Two Totally Ordered Subtypes of Duration** which do have a total order.

[XML Schema Part 2: Datatypes Second Edition] also states that the order relation on date and time datatypes is not a total order but a partial order because these datatypes may or may not have a timezone. This is handled as follows. If either operand to a comparison function on date or time values does not have an (explicit) timezone then, for the purpose of the operation, an implicit timezone, provided by the dynamic context Section C.2 Dynamic Context Components^XP, is assumed to be present as part of the value. This creates a total order for all date and time values.

An xs:dateTime can be considered to consist of seven components: year, month, day, hour, minute, second and timezone. For xs:dateTime six components: year, month, day, hour, minute and second are required and timezone is optional. For other date/time values, of the first six components, some are required and others must be absent or missing. Timezone is always optional. For example, for xs:date, the year, month and day components are required and hour, minute and second components must be absent; for xs:time the hour, minute and second components are required and year, month and day are missing; for xs:gDay, day is required and year, month, hour, minute and second are missing.

Values of the date/time datatypes xs:time, xs:gMonthDay, xs:gMonth, and xs:gDay, can be considered to represent a sequence of recurring time instants or time periods. An xs:time occurs every day. An xs:gMonth occurs every year. Comparison operators on these datatypes compare the starting instants of equivalent occurrences in the recurring series. These `xs:dateTime` values are calculated as described below.

Comparison operators on `xs:date`, `xs:gYearMonth` and `xs:gYear` compare their starting instants. These `xs:dateTime` values are calculated as described below.

The starting instant of an occurrence of a date/time value is an `xs:dateTime` calculated by filling in the missing components of the local value from a reference `xs:dateTime`. If the value filled in for a missing day component exceeds the maximum day value for the month, the last day of the month is used. Suppose, for example, that the reference `xs:dateTime` is `1972-12-31T00:00:00` and the `xs:date` value to be compared is `1993-03-31`. Filling in the time components from the reference `xs:dateTime` we get `1993-03-31T00:00:00` which is the starting instant of that day. Similarly, if the `xs:time` value `12:30:00` is to be compared, we fill in the missing components from the reference `xs:dateTime` and we get `1972-12-31T12:30:00` which is the time on that day. For an `xs:gYearMonth` value of `1976-02` we fill in the missing components, adjust for the last day in the month and get `1976-02-29T00:00:00`.

If the `xs:time` value written as `24:00:00` is to be compared, filling in the missing components gives `1972-12-31T00:00:00`, because `24:00:00` is an alternative representation of `00:00:00` (the lexical value `"24:00:00"` is converted to the time components {0,0,0} before the missing components are filled in). This has the consequence that when ordering `xs:time` values, `24:00:00` is considered to be earlier than `23:59:59`. However, when ordering `xs:dateTime` values, a time component of `24:00:00` is considered equivalent to `00:00:00` on the following day.

Note that the reference `xs:dateTime` does not have a timezone. The `timezone` component is never filled in from the reference `xs:dateTime`. In some cases, if the date/time value does not have a timezone, the implicit timezone from the dynamic context is used as the timezone.

> **Note:**
>
> This proposal uses the reference `xs:dateTime 1972-12-31T00:00:00` in the description of the comparison operators. Implementations are allowed to use other reference `xs:dateTime` values as long as they yield the same results. The reference `xs:dateTime` used must meet the following constraints: when it is used to supply components into `xs:gMonthDay` values, the year must allow for February 29 and so must be a leap year; when it is used to supply missing components into `xs:gDay` values, the month must allow for 31 days. Different reference `xs:dateTime` values may be used for different operators.

10.4.1 op:yearMonthDuration-less-than

```
op:yearMonthDuration-less-than($arg1 as xs:yearMonthDuration,
                               $arg2 as xs:yearMonthDuration) as xs:boolean
```

Summary: Returns `true` if and only if `$arg1` is less than `$arg2`. Returns `false` otherwise.

This function backs up the "lt" and "le" operators on `xs:yearMonthDuration` values.

10.4.2 op:yearMonthDuration-greater-than

```
op:yearMonthDuration-greater-than($arg1 as xs:yearMonthDuration,
                                  $arg2 as xs:yearMonthDuration) as xs:boolean
```

Summary: Returns `true` if and only if `$arg1` is greater than `$arg2`. Returns `false` otherwise.

This function backs up the "gt" and "ge" operators on `xs:yearMonthDuration` values.

10.4.3 op:dayTimeDuration-less-than

```
op:dayTimeDuration-less-than($arg1 as xs:dayTimeDuration,
                             $arg2 as xs:dayTimeDuration) as xs:boolean
```

Summary: Returns `true` if and only if `$arg1` is less than `$arg2`. Returns `false` otherwise.

This function backs up the "lt" and "le" operators on `xs:dayTimeDuration` values.

10.4.4 op:dayTimeDuration-greater-than

```
op:dayTimeDuration-greater-than($arg1 as xs:dayTimeDuration,
                                $arg2 as xs:dayTimeDuration) as xs:boolean
```

Summary: Returns `true` if and only if `$arg1` is greater than `$arg2`. Returns `false` otherwise.

This function backs up the "gt" and "ge" operators on `xs:dayTimeDuration` values.

10.4.5 op:duration-equal

```
op:duration-equal($arg1 as xs:duration, $arg2 as xs:duration) as xs:boolean
```

Summary: Returns `true` if and only if the `xs:yearMonthDuration` and the `xs:dayTimeDuration` components of `$arg1` and `$arg2` compare equal respectively. Returns `false` otherwise.

This function backs up the "eq" and "ne" operators on `xs:duration` values.

Note that this function, like any other, may be applied to arguments that are derived from the types given in the function signature, including the two subtypes `xs:dayTimeDuration` and `xs:yearMonthDuration`. With the exception of the zero-length duration, no instance of `xs:dayTimeDuration` can ever be equal to an instance of `xs:yearMonthDuration`.

The semantics of this function are:

```
xs:yearMonthDuration($arg1) div xs:yearMonthDuration('P1M')  eq
xs:yearMonthDuration($arg2) div xs:yearMonthDuration('P1M')
    and
xs:dayTimeDuration($arg1) div xs:dayTimeDuration('PT1S')  eq
xs:dayTimeDuration($arg2) div xs:dayTimeDuration('PT1S')
```

that is, the function returns `true` if the months and seconds values of the two durations are equal.

10.4.5.1 Examples

- `op:duration-equal(xs:duration("P1Y"), xs:duration("P12M"))` returns `true`.
- `op:duration-equal(xs:duration("PT24H"), xs:duration("P1D"))` returns `true`.
- `op:duration-equal(xs:duration("P1Y"), xs:duration("P365D"))` returns `false`.
- `op:duration-equal(xs:yearMonthDuration("P0Y"), xs:dayTimeDuration("P0D"))` returns `true`.
- `op:duration-equal(xs:yearMonthDuration("P1Y"), xs:dayTimeDuration("P365D"))` returns `false`.
- `op:duration-equal(xs:yearMonthDuration("P2Y"), xs:yearMonthDuration("P24M"))` returns `true`.
- `op:duration-equal(xs:dayTimeDuration("P10D"), xs:dayTimeDuration("PT240H"))` returns `true`.
- `op:duration-equal(xs:duration("P2Y0M0DT0H0M0S"), xs:yearMonthDuration("P24M"))` returns `true`.

- `op:duration-equal(xs:duration("P0Y0M10D"), xs:dayTimeDuration("PT240H"))` returns `true`.

10.4.6 op:dateTime-equal

```
op:dateTime-equal($arg1 as xs:dateTime, $arg2 as xs:dateTime) as xs:boolean
```

Summary: Returns `true` if and only if the value of `$arg1` is equal to the value of `$arg2` according to the algorithm defined in section 3.2.7.4 of [XML Schema Part 2: Datatypes Second Edition] "Order relation on dateTime" for `xs:dateTime` values with timezones. Returns `false` otherwise.

This function backs up the "eq", "ne", "le" and "ge" operators on `xs:dateTime` values.

10.4.6.1 Examples

Assume that the dynamic context provides an implicit timezone value of `-05:00`.

- `op:dateTime-equal(xs:dateTime("2002-04-02T12:00:00-01:00"), xs:dateTime("2002-04-02T17:00:00+04:00"))` returns `true`.
- `op:dateTime-equal(xs:dateTime("2002-04-02T12:00:00"), xs:dateTime("2002-04-02T23:00:00+06:00"))` returns `true`.
- `op:dateTime-equal(xs:dateTime("2002-04-02T12:00:00"), xs:dateTime("2002-04-02T17:00:00"))` returns `false`.
- `op:dateTime-equal(xs:dateTime("2002-04-02T12:00:00"), xs:dateTime("2002-04-02T12:00:00"))` returns `true`.
- `op:dateTime-equal(xs:dateTime("2002-04-02T23:00:00-04:00"), xs:dateTime("2002-04-03T02:00:00-01:00"))` returns `true`.
- `op:dateTime-equal(xs:dateTime("1999-12-31T24:00:00"), xs:dateTime("2000-01-01T00:00:00"))` returns `true`.
- `op:dateTime-equal(xs:dateTime("2005-04-04T24:00:00"), xs:dateTime("2005-04-04T00:00:00"))` returns `false`.

10.4.7 op:dateTime-less-than

```
op:dateTime-less-than($arg1 as xs:dateTime, $arg2 as xs:dateTime) as xs:boolean
```

Summary: Returns `true` if and only if the value of `$arg1` is less than the value of `$arg2` according to the algorithm defined in section 3.2.7.4 of [XML Schema Part 2: Datatypes Second Edition] "Order relation on dateTime" for `xs:dateTime` values with timezones. Returns `false` otherwise.

This function backs up the "lt" and "le" operators on `xs:dateTime` values.

10.4.8 op:dateTime-greater-than

```
op:dateTime-greater-than($arg1 as xs:dateTime,
                         $arg2 as xs:dateTime) as xs:boolean
```

Summary: Returns `true` if and only if the value of `$arg1` is greater than the value of `$arg2` according to the algorithm defined in section 3.2.7.4 of [XML Schema Part 2: Datatypes Second Edition] "Order relation on dateTime" for `xs:dateTime` values with timezones. Returns `false` otherwise.

This function backs up the "gt" and "ge" operators on `xs:dateTime` values.

10.4.9 op:date-equal

```
op:date-equal($arg1 as xs:date, $arg2 as xs:date) as xs:boolean
```

Summary: Returns `true` if and only if the starting instant of `$arg1` is equal to starting instant of `$arg2`. Returns `false` otherwise.

The starting instant of an `xs:date` is the `xs:dateTime` at time `00:00:00` on that date.

The two starting instants are compared using `op:dateTime-equal`.

This function backs up the "eq", "ne", "le" and "ge" operators on `xs:date` values.

10.4.9.1 Examples

- `op:date-equal(xs:date("2004-12-25Z"), xs:date("2004-12-25+07:00"))` returns `false`. The starting instants are `xs:dateTime("2004-12-25T00:00:00Z")` and `xs:dateTime("2004-12-25T00:00:00+07:00")`. These are normalized to `xs:dateTime("2004-12-25T00:00:00Z")` and `xs:dateTime("2004-12-24T17:00:00Z")`.
- `op:date-equal(xs:date("2004-12-25-12:00"), xs:date("2004-12-26+12:00"))` returns `true`.

10.4.10 op:date-less-than

```
op:date-less-than($arg1 as xs:date, $arg2 as xs:date) as xs:boolean
```

Summary: Returns `true` if and only if the starting instant of `$arg1` is less than the starting instant of `$arg2`. Returns `false` otherwise.

The starting instant of an `xs:date` is the `xs:dateTime` at time `00:00:00` on that date.

The two starting instants are compared using `op:dateTime-less-than`.

This function backs up the "lt" and "le" operators on `xs:date` values.

10.4.10.1 Examples

- `op:date-less-than(xs:date("2004-12-25Z"), xs:date("2004-12-25-05:00"))` returns `true`.
- `op:date-less-than(xs:date("2004-12-25-12:00"), xs:date("2004-12-26+12:00"))` returns `false`.

10.4.11 op:date-greater-than

```
op:date-greater-than($arg1 as xs:date, $arg2 as xs:date) as xs:boolean
```

Summary: Returns `true` if and only if the starting instant of `$arg1` is greater than the starting instant of `$arg2`. Returns `false` otherwise.

The starting instant of an `xs:date` is the `xs:dateTime` at time `00:00:00` on that date.

The two starting instants are compared using `op:dateTime-greater-than`.

This function backs up the "gt" and "ge" operators on `xs:date` values.

10.4.11.1 Examples

- `op:date-greater-than(xs:date("2004-12-25Z"), xs:date("2004-12-25+07:00"))` returns `true`.
- `op:date-greater-than(xs:date("2004-12-25-12:00"), xs:date("2004-12-26+12:00"))` returns `false`.

10.4.12 op:time-equal

```
op:time-equal($arg1 as xs:time, $arg2 as xs:time) as xs:boolean
```

Summary: Returns `true` if and only if the value of `$arg1` converted to an `xs:dateTime` using the date components from the reference `xs:dateTime` is equal to the value of `$arg2` converted to an `xs:dateTime` using the date components from the same reference `xs:dateTime`. Returns `false` otherwise.

The two `xs:dateTime` values are compared using `op:dateTime-equal`.

This function backs up the "eq", "ne", "le" and "ge" operators on `xs:time` values.

10.4.12.1 Examples

Assume that the date components from the reference `xs:dateTime` correspond to `1972-12-31`.

- `op:time-equal(xs:time("08:00:00+09:00"), xs:time("17:00:00-06:00"))` returns `false`. The `xs:dateTime`s calculated using the reference date components are `1972-12-31T08:00:00+09:00` and `1972-12-31T17:00:00-06:00`. These normalize to `1972-12-30T23:00:00Z` and `1972-12-31T23:00:00`.
- `op:time-equal(xs:time("21:30:00+10:30"), xs:time("06:00:00-05:00"))` returns `true`.
- `op:time-equal(xs:time("24:00:00+01:00"), xs:time("00:00:00+01:00"))` returns `true`. This not the result one might expect. For `xs:dateTime` values, a time of `24:00:00` is equivalent to `00:00:00` on the following day. For `xs:time`, the normalization from `24:00:00` to `00:00:00` happens before the `xs:time` is converted into an `xs:dateTime` for the purpose of the equality comparison. For `xs:time`, any operation on `24:00:00` produces the same result as the same operation on `00:00:00` because these are two different lexical representations of the same value.

10.4.13 op:time-less-than

```
op:time-less-than($arg1 as xs:time, $arg2 as xs:time) as xs:boolean
```

Summary: Returns `true` if and only if the value of `$arg1` converted to an `xs:dateTime` using the date components from the reference `xs:dateTime` is less than the normalized value of `$arg2` converted to an `xs:dateTime` using the date components from the same reference `xs:dateTime`. Returns `false` otherwise.

The two `xs:dateTime` values are compared using `op:dateTime-less-than`.

This function backs up the "lt" and "le" operators on `xs:time` values.

10.4.13.1 Examples

Assume that the dynamic context provides an implicit timezone value of `-05:00`.

- `op:time-less-than(xs:time("12:00:00"), xs:time("23:00:00+06:00"))` returns `false`.
- `op:time-less-than(xs:time("11:00:00"), xs:time("17:00:00Z"))` returns `true`.
- `op:time-less-than(xs:time("23:59:59"), xs:time("24:00:00"))` returns `false`.

10.4.14 op:time-greater-than

```
op:time-greater-than($arg1 as xs:time, $arg2 as xs:time) as xs:boolean
```

Summary: Returns `true` if and only if the value of `$arg1` converted to an `xs:dateTime` using the date components from the reference `xs:dateTime` is greater than the value of `$arg2` converted to an `xs:dateTime` using the date components from the same reference `xs:dateTime`. Returns `false` otherwise.

The two `xs:dateTime` values are compared using `op:dateTime-greater-than`.

This function backs up the "gt" and "ge" operators on `xs:time` values.

10.4.14.1 Examples

- `op:time-greater-than(xs:time("08:00:00+09:00"), xs:time("17:00:00-06:00"))` returns `false`.

10.4.15 op:gYearMonth-equal

```
op:gYearMonth-equal($arg1 as xs:gYearMonth,
                    $arg2 as xs:gYearMonth) as xs:boolean
```

Summary: Returns `true` if and only if the `xs:dateTime`s representing the starting instants of `$arg1` and `$arg2` compare equal. The starting instants of `$arg1` and `$arg2` are calculated by adding the missing components of `$arg1` and `$arg2` from the `xs:dateTime` template `xxxx-xx-ddT00:00:00` where `dd` represents the last day of the `month` component in `$arg1` or `$arg2`. Returns `false` otherwise.

The two `xs:dateTime` values representing the starting instants of `$arg1` and `$arg2` are compared using `op:dateTime-equal`.

This function backs up the "eq" and "ne" operators on `xs:gYearMonth` values.

10.4.15.1 Examples

Assume that the dynamic context provides an implicit timezone value of `-05:00`.

- `op:gYearMonth-equal(xs:gYearMonth("1976-02"), xs:gYearMonth("1976-03Z"))` returns `false`. The starting instants are `1972-02-29T00:00:00-05:00` and `1972-03-31T00:00:00Z`, respectively.
- `op:gYearMonth-equal(xs:gYearMonth("1976-03"), xs:gYearMonth("1976-03Z"))` returns `false`.

10.4.16 op:gYear-equal

```
op:gYear-equal($arg1 as xs:gYear, $arg2 as xs:gYear) as xs:boolean
```

Summary: Returns `true` if and only if the `xs:dateTime`s representing the starting instants of `$arg1` and `$arg2` compare equal. The starting instants of `$arg1` and `$arg2` are calculated by adding the missing components of `$arg1` and `$arg2` from a `xs:dateTime` template such as `xxxx-01-01T00:00:00`. Returns `false` otherwise.

The two `xs:dateTime` values representing the starting instants of `$arg1` and `$arg2` are compared using `op:dateTime-equal`.

This function backs up the "eq" and "ne" operators on `xs:gYear` values.

10.4.16.1 Examples

Assume that the dynamic context provides an implicit timezone value of `-05:00`. Assume, also, that the `xs:dateTime` template is `xxxx-01-01T00:00:00`.

- `op:gYear-equal(xs:gYear("2005-12:00"), xs:gYear("2005+12:00"))` returns `false`. The starting instants are `2005-01-01T00:00:00-12:00` and `2005-01-01T00:00:00+12:00`, respectively, and normalize to `2005-01-01T12:00:00Z` and `2004-12-31T12:00:00Z`.
- `op:gYear-equal(xs:gYear("1976-05:00"), xs:gYear("1976"))` returns `true`.

10.4.17 op:gMonthDay-equal

```
op:gMonthDay-equal($arg1 as xs:gMonthDay, $arg2 as xs:gMonthDay) as xs:boolean
```

Summary: Returns `true` if and only if the `xs:dateTime`s representing the starting instants of equivalent occurrences of `$arg1` and `$arg2` compare equal. The starting instants of equivalent occurrences of `$arg1` and `$arg2` are calculated by adding the missing components of `$arg1` and `$arg2` from an `xs:dateTime` template such as `1972-xx-xxT00:00:00`. Returns `false` otherwise.

The two `xs:dateTime` values representing the starting instants of equivalent occurrences of `$arg1` and `$arg2` are compared using `op:dateTime-equal`.

This function backs up the "eq" and "ne" operators on `xs:gMonthDay` values.

10.4.17.1 Examples

Assume that the dynamic context provides an implicit timezone value of `-05:00`. Assume, also, that the `xs:dateTime` template is `1976-xx-xxT00:00:00`.

- `op:gMonthDay-equal(xs:gMonthDay("--12-25-14:00"), xs:gMonthDay("--12-26+10:00"))` returns `true`. The starting instants are `1976-12-25T00:00:00-14:00` and `1976-12-26T00:00:00+10:00`, respectively, and normalize to `1976-12-25T14:00:00Z` and `1976-12-25T14:00:00Z`.
- `op:gMonthDay-equal(xs:gMonthDay("--12-25"), xs:gMonthDay("--12-26Z"))` returns `false`.

10.4.18 op:gMonth-equal

```
op:gMonth-equal($arg1 as xs:gMonth, $arg2 as xs:gMonth) as xs:boolean
```

Summary: Returns `true` if and only if the `xs:dateTime`s representing the starting instants of equivalent occurrences of `$arg1` and `$arg2` compare equal. The starting instants of equivalent occurrences of `$arg1` and `$arg2` are calculated by adding the missing components of `$arg1` and `$arg2` from an `xs:dateTime` template such as `1972-xx-ddT00:00:00` where `dd` represents the last day of the month component in `$arg1` or `$arg2`. Returns `false` otherwise.

The two `xs:dateTime` values representing the starting instants of equivalent occurrences of `$arg1` and `$arg2` are compared using `op:dateTime-equal`.

This function backs up the "eq" and "ne" operators on `xs:gMonth` values.

10.4.18.1 Examples

Assume that the dynamic context provides an implicit timezone value of `-05:00`. Assume, also, that the `xs:dateTime` template is `1972-xx-29T00:00:00`.

- `op:gMonth-equal(xs:gMonth("--12-14:00"), xs:gMonth("--12+10:00"))` returns `false`. The starting instants are `1972-12-29T00:00:00-14:00` and `1972-12-29T00:00:00+10:00`, respectively, and normalize to `1972-12-29T14:00:00Z` and `1972-12-28T14:00:00Z`.
- `op:gMonth-equal(xs:gMonth("--12"), xs:gMonth("--12Z"))` returns `false`.

10.4.19 op:gDay-equal

```
op:gDay-equal($arg1 as xs:gDay, $arg2 as xs:gDay) as xs:boolean
```

Summary: Returns `true` if and only if the `xs:dateTime`s representing the starting instants of equivalent occurrences of `$arg1` and `$arg2` compare equal. The starting instants of equivalent occurrences of `$arg1` and `$arg2` are calculated by adding the missing components of `$arg1` and `$arg2` from an `xs:dateTime` template such as `1972-12-xxT00:00:00`. Returns `false` otherwise.

The two `xs:dateTime` values representing the starting instants of equivalent occurrences of `$arg1` and `$arg2` are compared using `op:dateTime-equal`.

This function backs up the "eq" and "ne" operators on `xs:gDay` values.

10.4.19.1 Examples

Assume that the dynamic context provides an implicit timezone value of `-05:00`. Assume, also, that the `xs:dateTime` template is `1976-12-xxT00:00:00`.

- `op:gDay-equal(xs:gDay("---25-14:00"), xs:gDay("---25+10:00"))` returns `false`. The starting instants are `1972-12-25T00:00:00-14:00` and `1972-12-25T00:00:00+10:00`, respectively, and normalize to `1972-12-25T14:00:00Z` and `1972-12-24T14:00:00Z`.
- `op:gDay-equal(xs:gDay("---12"), xs:gDay("---12Z"))` returns `false`.

10.5 Component Extraction Functions on Durations, Dates and Times

The duration, date and time datatypes may be considered to be composite datatypes in that they contain distinct properties or components. The extraction functions specified below extract a single component from a duration, date or time value. For the date/time datatypes the local value is used. For `xs:duration` and its subtypes, including the two subtypes `xs:yearMonthDuration` and `xs:dayTimeDuration`, the components are normalized: this means that the seconds and minutes components will always be less than 60, the hours component less than 24, and the months component less than 12.

Function	Meaning
`fn:years-from-duration`	Returns the year component of an `xs:duration` value.
`fn:months-from-duration`	Returns the months component of an `xs:duration` value.

Function	Meaning
fn:days-from-duration	Returns the days component of an `xs:duration` value.
fn:hours-from-duration	Returns the hours component of an `xs:duration` value.
fn:minutes-from-duration	Returns the minutes component of an `xs:duration` value.
fn:seconds-from-duration	Returns the seconds component of an `xs:duration` value.
fn:year-from-dateTime	Returns the year from an `xs:dateTime` value.
fn:month-from-dateTime	Returns the month from an `xs:dateTime` value.
fn:day-from-dateTime	Returns the day from an `xs:dateTime` value.
fn:hours-from-dateTime	Returns the hours from an `xs:dateTime` value.
fn:minutes-from-dateTime	Returns the minutes from an `xs:dateTime` value.
fn:seconds-from-dateTime	Returns the seconds from an `xs:dateTime` value.
fn:timezone-from-dateTime	Returns the timezone from an `xs:dateTime` value.
fn:year-from-date	Returns the year from an `xs:date` value.
fn:month-from-date	Returns the month from an `xs:date` value.
fn:day-from-date	Returns the day from an `xs:date` value.
fn:timezone-from-date	Returns the timezone from an `xs:date` value.
fn:hours-from-time	Returns the hours from an `xs:time` value.
fn:minutes-from-time	Returns the minutes from an `xs:time` value.
fn:seconds-from-time	Returns the seconds from an `xs:time` value.
fn:timezone-from-time	Returns the timezone from an `xs:time` value.

10.5.1 fn:years-from-duration

```
fn:years-from-duration($arg as xs:duration?) as xs:integer?
```

Summary: Returns an `xs:integer` representing the years component in the value of `$arg`. The result is obtained by casting `$arg` to an `xs:yearMonthDuration` (see **17.1.4 Casting to duration types**) and then computing the years component as described in **10.3.1.3 Canonical representation**.

The result may be negative.

If `$arg` is an `xs:dayTimeDuration` returns 0.

If `$arg` is the empty sequence, returns the empty sequence.

10.5.1.1 Examples

- `fn:years-from-duration(xs:yearMonthDuration("P20Y15M"))` returns `21`.
- `fn:years-from-duration(xs:yearMonthDuration("-P15M"))` returns `-1`.
- `fn:years-from-duration(xs:dayTimeDuration("-P2DT15H"))` returns `0`.

10.5.2 fn:months-from-duration

```
fn:months-from-duration($arg as xs:duration?) as xs:integer?
```

Summary: Returns an `xs:integer` representing the months component in the value of `$arg`. The result is obtained by casting `$arg` to an `xs:yearMonthDuration` (see **17.1.4 Casting to duration** types) and then computing the months component as described in 10.3.1.3 Canonical representation.

The result may be negative.

If `$arg` is an `xs:dayTimeDuration` returns 0.

If `$arg` is the empty sequence, returns the empty sequence.

10.5.2.1 Examples

- `fn:months-from-duration(xs:yearMonthDuration("P20Y15M"))` returns `3`.
- `fn:months-from-duration(xs:yearMonthDuration("-P20Y18M"))` returns `-6`.
- `fn:months-from-duration(xs:dayTimeDuration("-P2DT15H0M0S"))` returns `0`.

10.5.3 fn:days-from-duration

```
fn:days-from-duration($arg as xs:duration?) as xs:integer?
```

Summary: Returns an `xs:integer` representing the days component in the value of `$arg`. The result is obtained by casting `$arg` to an `xs:dayTimeDuration` (see 17.1.4 Casting to duration types) and then computing the days component as described in 10.3.2.3 Canonical representation.

The result may be negative.

If `$arg` is an `xs:yearMonthDuration` returns 0.

If `$arg` is the empty sequence, returns the empty sequence.

10.5.3.1 Examples

- `fn:days-from-duration(xs:dayTimeDuration("P3DT10H"))` returns `3`.
- `fn:days-from-duration(xs:dayTimeDuration("P3DT55H"))` returns `5`.
- `fn:days-from-duration(xs:yearMonthDuration("P3Y5M"))` returns `0`.

10.5.4 fn:hours-from-duration

```
fn:hours-from-duration($arg as xs:duration?) as xs:integer?
```

Summary: Returns an `xs:integer` representing the hours component in the value of `$arg`. The result is obtained by casting `$arg` to an `xs:dayTimeDuration` (see 17.1.4 Casting to duration types) and then computing the hours component as described in 10.3.2.3 Canonical representation.

The result may be negative.

If `$arg` is an `xs:yearMonthDuration` returns 0.

If `$arg` is the empty sequence, returns the empty sequence.

10.5.4.1 Examples

- `fn:hours-from-duration(xs:dayTimeDuration("P3DT10H"))` returns `10`.

- `fn:hours-from-duration(xs:dayTimeDuration("P3DT12H32M12S"))` returns `12`.
- `fn:hours-from-duration(xs:dayTimeDuration("PT123H"))` returns `3`.
- `fn:hours-from-duration(xs:dayTimeDuration("-P3DT10H"))` returns `-10`.

10.5.5 fn:minutes-from-duration

```
fn:minutes-from-duration($arg as xs:duration?) as xs:integer?
```

Summary: Returns an `xs:integer` representing the minutes component in the value of `$arg`. The result is obtained by casting `$arg` to an `xs:dayTimeDuration` (see **17.1.4 Casting to duration types**) and then computing the minutes component as described in **10.3.2.3 Canonical representation**.

The result may be negative.

If `$arg` is an `xs:yearMonthDuration` returns 0.

If `$arg` is the empty sequence, returns the empty sequence.

10.5.5.1 Examples

- `fn:minutes-from-duration(xs:dayTimeDuration("P3DT10H"))` returns `0`.
- `fn:minutes-from-duration(xs:dayTimeDuration("-P5DT12H30M"))` returns `-30`.

10.5.6 fn:seconds-from-duration

```
fn:seconds-from-duration($arg as xs:duration?) as xs:decimal?
```

Summary: Returns an `xs:decimal` representing the seconds component in the value of `$arg`. The result is obtained by casting `$arg` to an `xs:dayTimeDuration` (see **17.1.4 Casting to duration types**) and then computing the seconds component as described in **10.3.2.3 Canonical representation**.

The result may be negative.

If `$arg` is an `xs:yearMonthDuration` returns 0.

If `$arg` is the empty sequence, returns the empty sequence.

10.5.6.1 Examples

- `fn:seconds-from-duration(xs:dayTimeDuration("P3DT10H12.5S"))` returns `12.5`.
- `fn:seconds-from-duration(xs:dayTimeDuration("-PT256S"))` returns `-16.0`.

10.5.7 fn:year-from-dateTime

```
fn:year-from-dateTime($arg as xs:dateTime?) as xs:integer?
```

Summary: Returns an `xs:integer` representing the year component in the localized value of `$arg`. The result may be negative.

If `$arg` is the empty sequence, returns the empty sequence.

10.5.7.1 Examples

- `fn:year-from-dateTime(xs:dateTime("1999-05-31T13:20:00-05:00"))` returns `1999`.
- `fn:year-from-dateTime(xs:dateTime("1999-05-31T21:30:00-05:00"))` returns `1999`.
- `fn:year-from-dateTime(xs:dateTime("1999-12-31T19:20:00"))` returns `1999`.
- `fn:year-from-dateTime(xs:dateTime("1999-12-31T24:00:00"))` returns `2000`.

10.5.8 fn:month-from-dateTime

```
fn:month-from-dateTime($arg as xs:dateTime?) as xs:integer?
```

Summary: Returns an `xs:integer` between 1 and 12, both inclusive, representing the month component in the localized value of `$arg`.

If `$arg` is the empty sequence, returns the empty sequence.

10.5.8.1 Examples

- `fn:month-from-dateTime(xs:dateTime("1999-05-31T13:20:00-05:00"))` returns `5`.
- `fn:month-from-dateTime(xs:dateTime("1999-12-31T19:20:00-05:00"))` returns `12`.
- `fn:month-from-dateTime(fn:adjust-dateTime-to-timezone(xs:dateTime("1999-12-31T19:20:00-05:00"), xs:dayTimeDuration("PT0S")))` returns `1`.

10.5.9 fn:day-from-dateTime

```
fn:day-from-dateTime($arg as xs:dateTime?) as xs:integer?
```

Summary: Returns an `xs:integer` between 1 and 31, both inclusive, representing the day component in the localized value of `$arg`.

If `$arg` is the empty sequence, returns the empty sequence.

10.5.9.1 Examples

- `fn:day-from-dateTime(xs:dateTime("1999-05-31T13:20:00-05:00"))` returns `31`.
- `fn:day-from-dateTime(xs:dateTime("1999-12-31T20:00:00-05:00"))` returns `31`.
- `fn:day-from-dateTime(fn:adjust-dateTime-to-timezone(xs:dateTime("1999-12-31T19:20:00-05:00"), xs:dayTimeDuration("PT0S")))` returns `1`.

10.5.10 fn:hours-from-dateTime

```
fn:hours-from-dateTime($arg as xs:dateTime?) as xs:integer?
```

Summary: Returns an `xs:integer` between 0 and 23, both inclusive, representing the hours component in the localized value of `$arg`.

If `$arg` is the empty sequence, returns the empty sequence.

10.5.10.1 Examples

- `fn:hours-from-dateTime(xs:dateTime("1999-05-31T08:20:00-05:00"))` returns `8`.
- `fn:hours-from-dateTime(xs:dateTime("1999-12-31T21:20:00-05:00"))` returns `21`.
- `fn:hours-from-dateTime(fn:adjust-dateTime-to-timezone(xs:dateTime("1999-12-31T21:20:00-05:00"), xs:dayTimeDuration("PT0S")))` returns `2`.
- `fn:hours-from-dateTime(xs:dateTime("1999-12-31T12:00:00"))` returns `12`.
- `fn:hours-from-dateTime(xs:dateTime("1999-12-31T24:00:00"))` returns `0`.

10.5.11 fn:minutes-from-dateTime

```
fn:minutes-from-dateTime($arg as xs:dateTime?) as xs:integer?
```

Summary: Returns an `xs:integer` value between 0 and 59, both inclusive, representing the minute component in the localized value of `$arg`.

If `$arg` is the empty sequence, returns the empty sequence.

10.5.11.1 Examples

- `fn:minutes-from-dateTime(xs:dateTime("1999-05-31T13:20:00-05:00"))` returns `20` .
- `fn:minutes-from-dateTime(xs:dateTime("1999-05-31T13:30:00+05:30"))` returns `30` .

10.5.12 fn:seconds-from-dateTime

```
fn:seconds-from-dateTime($arg as xs:dateTime?) as xs:decimal?
```

Summary: Returns an `xs:decimal` value greater than or equal to zero and less than 60, representing the seconds and fractional seconds in the localized value of `$arg`.

If `$arg` is the empty sequence, returns the empty sequence.

10.5.12.1 Examples

- `fn:seconds-from-dateTime(xs:dateTime("1999-05-31T13:20:00-05:00"))` returns `0`.

10.5.13 fn:timezone-from-dateTime

```
fn:timezone-from-dateTime($arg as xs:dateTime?) as xs:dayTimeDuration?
```

Summary: Returns the timezone component of `$arg` if any. If `$arg` has a timezone component, then the result is an `xs:dayTimeDuration` that indicates deviation from UTC; its value may range from +14:00 to -14:00 hours, both inclusive. Otherwise, the result is the empty sequence.

If `$arg` is the empty sequence, returns the empty sequence.

10.5.13.1 Examples

- `fn:timezone-from-dateTime(xs:dateTime("1999-05-31T13:20:00-05:00"))` returns the `xs:dayTimeDuration` whose value is `-PT5H`.

- `fn:timezone-from-dateTime(xs:dateTime("2000-06-12T13:20:00Z"))` returns the `xs:dayTimeDuration` whose value is `PT0S`.
- `fn:timezone-from-dateTime(xs:dateTime("2004-08-27T00:00:00"))` returns `()`.

10.5.14 fn:year-from-date

`fn:year-from-date``($arg as` *`xs:date?`*`) as` *`xs:integer?`*

Summary: Returns an `xs:integer` representing the year in the localized value of `$arg`. The value may be negative.

If `$arg` is the empty sequence, returns the empty sequence.

10.5.14.1 Examples

- `fn:year-from-date(xs:date("1999-05-31"))` returns `1999`.
- `fn:year-from-date(xs:date("2000-01-01+05:00"))` returns `2000`.

10.5.15 fn:month-from-date

`fn:month-from-date``($arg as` *`xs:date?`*`) as` *`xs:integer?`*

Summary: Returns an `xs:integer` between 1 and 12, both inclusive, representing the month component in the localized value of `$arg`.

If `$arg` is the empty sequence, returns the empty sequence.

10.5.15.1 Examples

- `fn:month-from-date(xs:date("1999-05-31-05:00"))` returns `5` .
- `fn:month-from-date(xs:date("2000-01-01+05:00"))` returns `1`.

10.5.16 fn:day-from-date

`fn:day-from-date``($arg as` *`xs:date?`*`) as` *`xs:integer?`*

Summary: Returns an `xs:integer` between 1 and 31, both inclusive, representing the day component in the localized value of `$arg`.

If `$arg` is the empty sequence, returns the empty sequence.

10.5.16.1 Examples

- `fn:day-from-date(xs:date("1999-05-31-05:00"))` returns `31`.
- `fn:day-from-date(xs:date("2000-01-01+05:00"))` returns `1`.

10.5.17 fn:timezone-from-date

`fn:timezone-from-date``($arg as` *`xs:date?`*`) as` *`xs:dayTimeDuration?`*

Summary: Returns the timezone component of `$arg` if any. If `$arg` has a timezone component, then the result is an `xs:dayTimeDuration` that indicates deviation from UTC; its value may range from +14:00 to -14:00 hours, both inclusive. Otherwise, the result is the empty sequence.

If `$arg` is the empty sequence, returns the empty sequence.

10.5.17.1 Examples

- `fn:timezone-from-date(xs:date("1999-05-31-05:00"))` returns the `xs:dayTimeDuration` whose value is `-PT5H`.
- `fn:timezone-from-date(xs:date("2000-06-12Z"))` returns the `xs:dayTimeDuration` with value `PT0S`.

10.5.18 fn:hours-from-time

```
fn:hours-from-time($arg as xs:time?) as xs:integer?
```

Summary: Returns an `xs:integer` between 0 and 23, both inclusive, representing the value of the hours component in the localized value of `$arg`.

If `$arg` is the empty sequence, returns the empty sequence.

10.5.18.1 Examples

Assume that the dynamic context provides an implicit timezone value of `-05:00`.

- `fn:hours-from-time(xs:time("11:23:00"))` returns `11`.
- `fn:hours-from-time(xs:time("21:23:00"))` returns `21`.
- `fn:hours-from-time(xs:time("01:23:00+05:00"))` returns `1`.
- `fn:hours-from-time(fn:adjust-time-to-timezone(xs:time("01:23:00+05:00"), xs:dayTimeDuration("PT0S")))` returns `20`.
- `fn:hours-from-time(xs:time("24:00:00"))` returns `0`.

10.5.19 fn:minutes-from-time

```
fn:minutes-from-time($arg as xs:time?) as xs:integer?
```

Summary: Returns an `xs:integer` value between 0 and 59, both inclusive, representing the value of the minutes component in the localized value of `$arg`.

If `$arg` is the empty sequence, returns the empty sequence.

10.5.19.1 Examples

- `fn:minutes-from-time(xs:time("13:00:00Z"))` returns `0` .

10.5.20 fn:seconds-from-time

```
fn:seconds-from-time($arg as xs:time?) as xs:decimal?
```

Summary: Returns an `xs:decimal` value greater than or equal to zero and less than 60, representing the seconds and fractional seconds in the localized value of `$arg`.

If `$arg` is the empty sequence, returns the empty sequence.

10.5.20.1 Examples

- `fn:seconds-from-time(xs:time("13:20:10.5"))` returns `10.5`.

10.5.21 fn:timezone-from-time

```
fn:timezone-from-time($arg as xs:time?) as xs:dayTimeDuration?
```

Summary: Returns the timezone component of `$arg` if any. If `$arg` has a timezone component, then the result is an `xs:dayTimeDuration` that indicates deviation from UTC; its value may range from +14:00 to -14:00 hours, both inclusive. Otherwise, the result is the empty sequence.

If `$arg` is the empty sequence, returns the empty sequence.

10.5.21.1 Examples

- `fn:timezone-from-time(xs:time("13:20:00-05:00"))` returns `xs:dayTimeDuration` whose value is `-PT5H`.
- `fn:timezone-from-time(xs:time("13:20:00"))` returns `()`.

10.6 Arithmetic Operators on Durations

Function	Meaning
`op:add-yearMonthDurations`	Adds two `xs:yearMonthDuration`s. Returns an `xs:yearMonthDuration`.
`op:subtract-yearMonthDurations`	Subtracts one `xs:yearMonthDuration` from another. Returns an `xs:yearMonthDuration`.
`op:multiply-yearMonthDuration`	Multiplies a `xs:yearMonthDuration` by an `xs:double`. Returns an `xs:yearMonthDuration`.
`op:divide-yearMonthDuration`	Divides an `xs:yearMonthDuration` by an `xs:double`. Returns an `xs:yearMonthDuration`.
`op:divide-yearMonthDuration-by-yearMonthDuration`	Divides an `xs:yearMonthDuration` by an `xs:yearMonthDuration`. Returns an `xs:decimal`.
`op:add-dayTimeDurations`	Adds two `xs:dayTimeDurations`. Returns an `xs:dayTimeDuration`.
`op:subtract-dayTimeDurations`	Subtracts one `xs:dayTimeDuration` from another. Returns an `xs:dayTimeDuration`.
`op:multiply-dayTimeDuration`	Multiplies an `xs:dayTimeDuration` by a `xs:double`. Returns an `xs:dayTimeDuration`.
`op:divide-dayTimeDuration`	Divides an `xs:dayTimeDuration` by an `xs:double`. Returns an `xs:dayTimeDuration`.
`op:divide-dayTimeDuration-by-dayTimeDuration`	Divides an `xs:dayTimeDuration` by an `xs:dayTimeDuration`. Returns an `xs:decimal`.

10.6.1 op:add-yearMonthDurations

```
op:add-yearMonthDurations($arg1 as xs:yearMonthDuration,
                          $arg2 as xs:yearMonthDuration) as xs:yearMonthDuration
```

Summary: Returns the result of adding the value of `$arg1` to the value of `$arg2`. Backs up the "+" operator on `xs:yearMonthDuration` values.

10.6.1.1 Examples

- `op:add-yearMonthDurations(xs:yearMonthDuration("P2Y11M"), xs:yearMonthDuration("P3Y3M"))` returns a `xs:yearMonthDuration` value corresponding to 6 years and 2 months.

10.6.2 op:subtract-yearMonthDurations

```
op:subtract-yearMonthDurations($arg1 as xs:yearMonthDuration,
                               $arg2 as xs:yearMonthDuration) as xs:yearMonthDuration
```

Summary: Returns the result of subtracting the value of `$arg2` from the value of `$arg1`. Backs up the "-" operator on `xs:yearMonthDuration` values.

10.6.2.1 Examples

- `op:subtract-yearMonthDurations(xs:yearMonthDuration("P2Y11M"), xs:yearMonthDuration("P3Y3M"))` returns a `xs:yearMonthDuration` value corresponding to negative 4 months.

10.6.3 op:multiply-yearMonthDuration

```
op:multiply-yearMonthDuration($arg1 as xs:yearMonthDuration,
                              $arg2 as xs:double) as xs:yearMonthDuration
```

Summary: Returns the result of multiplying the value of `$arg1` by `$arg2`. The result is rounded to the nearest month.

The result is the `xs:yearMonthDuration` whose length in months is equal to the result of applying the `fn:round` function to the value obtained by multiplying the length in months of `$arg1` by the value of `$arg2`.

If `$arg2` is positive or negative zero, the result is a zero-length duration. If `$arg2` is positive or negative infinity, the result overflows and is handled as discussed in **10.1.1 Limits and Precision**. If `$arg2` is `NaN` an error is raised [err:FOCA0005]

Backs up the "*" operator on `xs:yearMonthDuration` values.

10.6.3.1 Examples

- `op:multiply-yearMonthDuration(xs:yearMonthDuration("P2Y11M"), 2.3)` returns a `xs:yearMonthDuration` value corresponding to 6 years and 9 months.

10.6.4 op:divide-yearMonthDuration

```
op:divide-yearMonthDuration($arg1 as xs:yearMonthDuration,
                            $arg2 as xs:double) as xs:yearMonthDuration
```

Summary: Returns the result of dividing the value of `$arg1` by `$arg2`. The result is rounded to the nearest month.

The result is the `xs:yearMonthDuration` whose length in months is equal to the result of applying the `fn:round` function to the value obtained by dividing the length in months of `$arg1` by the value of `$arg2`.

If `$arg2` is positive or negative infinity, the result is a zero-length duration. If `$arg2` is positive or negative zero, the result overflows and is handled as discussed in **10.1.1 Limits and Precision**. If `$arg2` is `NaN` an error is raised [err:FOCA0005]

Backs up the "div" operator on `xs:yearMonthDuration` and numeric values.

10.6.4.1 Examples

- `op:divide-yearMonthDuration(xs:yearMonthDuration("P2Y11M"), 1.5)` returns a `xs:yearMonthDuration` value corresponding to 1 year and 11 months.

10.6.5 op:divide-yearMonthDuration-by-yearMonthDuration

```
op:divide-yearMonthDuration-by-yearMonthDuration($arg1 as xs:yearMonthDuration,
                                                 $arg2 as xs:yearMonthDuration) as xs:decimal
```

Summary: Returns the result of dividing the value of `$arg1` by `$arg2`. Since the values of both operands are integers, the semantics of the division is identical to `op:numeric-divide` with `xs:integer` operands.

Backs up the "div" operator on `xs:yearMonthDuration` values.

10.6.5.1 Examples

- `op:divide-yearMonthDuration-by-yearMonthDuration(xs:yearMonthDuration("P3Y4M"), xs:yearMonthDuration("-P1Y4M"))` returns `-2.5`.

10.6.6 op:add-dayTimeDurations

```
op:add-dayTimeDurations($arg1 as xs:dayTimeDuration,
                        $arg2 as xs:dayTimeDuration) as xs:dayTimeDuration
```

Summary: Returns the result of adding the value of `$arg1` to the value of `$arg2`. Backs up the "+" operator on `xs:dayTimeDuration` values.

10.6.6.1 Examples

- `op:add-dayTimeDurations(xs:dayTimeDuration("P2DT12H5M"), xs:dayTimeDuration("P5DT12H"))` returns a `xs:dayTimeDuration` value corresponding to 8 days and 5 minutes.

10.6.7 op:subtract-dayTimeDurations

```
op:subtract-dayTimeDurations($arg1 as xs:dayTimeDuration,
                             $arg2 as xs:dayTimeDuration) as xs:dayTimeDuration
```

Summary: Returns the result of subtracting the value of `$arg2` from the value of `$arg1`. Backs up the "-" operator on `xs:dayTimeDuration` values.

10.6.7.1 Examples

- `op:subtract-dayTimeDurations(xs:dayTimeDuration("P2DT12H"), xs:dayTimeDuration("P1DT10H30M"))` returns a `xs:dayTimeDuration` value corresponding to 1 day, 1 hour and 30 minutes.

10.6.8 op:multiply-dayTimeDuration

```
op:multiply-dayTimeDuration($arg1 as xs:dayTimeDuration,
                            $arg2 as xs:double) as xs:dayTimeDuration
```

Summary: Returns the result of multiplying the value of `$arg1` by `$arg2`.

If `$arg2` is positive or negative zero, the result is a zero-length duration. If `$arg2` is positive or negative infinity, the result overflows and is handled as discussed in **10.1.1 Limits and Precision**. If `$arg2` is `NaN` an error is raised [err:FOCA0005]

Backs up the "*" operator on `xs:dayTimeDuration` values.

10.6.8.1 Examples

- `op:multiply-dayTimeDuration(xs:dayTimeDuration("PT2H10M"), 2.1)` returns a `xs:dayTimeDuration` value corresponding to 4 hours and 33 minutes.

10.6.9 op:divide-dayTimeDuration

```
op:divide-dayTimeDuration($arg1 as xs:dayTimeDuration,
                          $arg2 as xs:double) as xs:dayTimeDuration
```

Summary: Returns the result of dividing the value of `$arg1` by `$arg2`.

If `$arg2` is positive or negative infinity, the result is a zero-length duration. If `$arg2` is positive or negative zero, the result overflows and is handled as discussed in **10.1.1 Limits and Precision**. If `$arg2` is `NaN` an error is raised [err:FOCA0005]

Backs up the "div" operator on `xs:dayTimeDuration` values.

10.6.9.1 Examples

- `op:divide-dayTimeDuration(xs:dayTimeDuration("P1DT2H30M10.5S"), 1.5)` returns an `xs:dayTimeDuration` value corresponding to 17 hours, 40 minutes and 7 seconds.

10.6.10 op:divide-dayTimeDuration-by-dayTimeDuration

```
op:divide-dayTimeDuration-by-dayTimeDuration($arg1 as xs:dayTimeDuration,
                                   $arg2 as xs:dayTimeDuration) as xs:decimal
```

Summary: Returns the result of dividing the value of `$arg1` by `$arg2`. Since the values of both operands are decimals, the semantics of the division is identical to `op:numeric-divide` with `xs:decimal` operands.

Backs up the "div" operator on `xs:dayTimeDuration` values.

10.6.10.1 Examples

- `op:divide-dayTimeDuration-by-dayTimeDuration(xs:dayTimeDuration("P2DT53M11S"), xs:dayTimeDuration("P1DT10H"))` returns `1.4378349...`

10.7 Timezone Adjustment Functions on Dates and Time Values

Function	Meaning
`fn:adjust-dateTime-to-timezone`	Adjusts an `xs:dateTime` value to a specific timezone, or to no timezone at all.
`fn:adjust-date-to-timezone`	Adjusts an `xs:date` value to a specific timezone, or to no timezone at all.
`fn:adjust-time-to-timezone`	Adjusts an `xs:time` value to a specific timezone, or to no timezone at all.

These functions adjust the timezone component of an `xs:dateTime`, `xs:date` or `xs:time` value. The `$timezone` argument to these functions is defined as an `xs:dayTimeDuration` but must be a valid timezone value.

10.7.1 fn:adjust-dateTime-to-timezone

```
fn:adjust-dateTime-to-timezone($arg as xs:dateTime?) as xs:dateTime?
```

```
fn:adjust-dateTime-to-timezone($arg      as xs:dateTime?,
                               $timezone as xs:dayTimeDuration?) as xs:dateTime?
```

Summary: Adjusts an `xs:dateTime` value to a specific timezone, or to no timezone at all. If `$timezone` is the empty sequence, returns an `xs:dateTime` without a timezone. Otherwise, returns an `xs:dateTime` with a timezone.

If `$timezone` is not specified, then `$timezone` is the value of the implicit timezone in the dynamic context.

If `$arg` is the empty sequence, then the result is the empty sequence.

A dynamic error is raised [err:FODT0003] if `$timezone` is less than `-PT14H` or greater than `PT14H` or if does not contain an integral number of minutes.

If `$arg` does not have a timezone component and `$timezone` is the empty sequence, then the result is `$arg`.

If `$arg` does not have a timezone component and `$timezone` is not the empty sequence, then the result is `$arg` with `$timezone` as the timezone component.

If `$arg` has a timezone component and `$timezone` is the empty sequence, then the result is the localized value of `$arg` without its timezone component.

If `$arg` has a timezone component and `$timezone` is not the empty sequence, then the result is an `xs:dateTime` value with a timezone component of `$timezone` that is equal to `$arg`.

10.7.1.1 Examples

Assume the dynamic context provides an implicit timezone of `-05:00 (-PT5H0M)`.

```
let $tz := xs:dayTimeDuration("-PT10H")
```

- `fn:adjust-dateTime-to-timezone(xs:dateTime("2002-03-07T10:00:00"))` returns `2002-03-07T10:00:00-05:00`
- `fn:adjust-dateTime-to-timezone(xs:dateTime("2002-03-07T10:00:00-07:00"))` returns `2002-03-07T12:00:00-05:00`
- `fn:adjust-dateTime-to-timezone(xs:dateTime("2002-03-07T10:00:00"), $tz)` returns `2002-03-07T10:00:00-10:00`
- `fn:adjust-dateTime-to-timezone(xs:dateTime("2002-03-07T10:00:00-07:00"), $tz)` returns `2002-03-07T07:00:00-10:00`
- `fn:adjust-dateTime-to-timezone(xs:dateTime("2002-03-07T10:00:00-07:00"), xs:dayTimeDuration("PT10H"))` returns `2002-03-08T03:00:00+10:00`
- `fn:adjust-dateTime-to-timezone(xs:dateTime("2002-03-07T00:00:00+01:00"), xs:dayTimeDuration("-PT8H"))` returns `2002-03-06T15:00:00-08:00`
- `fn:adjust-dateTime-to-timezone(xs:dateTime("2002-03-07T10:00:00"), ())` returns `2002-03-07T10:00:00`
- `fn:adjust-dateTime-to-timezone(xs:dateTime("2002-03-07T10:00:00-07:00"), ())` returns `2002-03-07T10:00:00`

10.7.2 fn:adjust-date-to-timezone

```
fn:adjust-date-to-timezone($arg as xs:date?) as xs:date?
```

```
fn:adjust-date-to-timezone($arg      as xs:date?,
                           $timezone as xs:dayTimeDuration?) as xs:date?
```

Summary: Adjusts an `xs:date` value to a specific timezone, or to no timezone at all. If `$timezone` is the empty sequence, returns an `xs:date` without a timezone. Otherwise, returns an `xs:date` with a timezone. For purposes of timezone adjustment, an `xs:date` is treated as an `xs:dateTime` with time `00:00:00`.

If `$timezone` is not specified, then `$timezone` is the value of the implicit timezone in the dynamic context.

If `$arg` is the empty sequence, then the result is the empty sequence.

A dynamic error is raised [err:FODT0003] if `$timezone` is less than `-PT14H` or greater than `PT14H` or if does not contain an integral number of minutes.

If `$arg` does not have a timezone component and `$timezone` is the empty sequence, then the result is the value of `$arg`.

If `$arg` does not have a timezone component and `$timezone` is not the empty sequence, then the result is `$arg` with `$timezone` as the timezone component.

If `$arg` has a timezone component and `$timezone` is the empty sequence, then the result is the localized value of `$arg` without its timezone component.

If `$arg` has a timezone component and `$timezone` is not the empty sequence, then:

- Let `$srcdt` be an `xs:dateTime` value, with `00:00:00` for the time component and date and timezone components that are the same as the date and timezone components of `$arg`.
- Let `$r` be the result of evaluating [`fn:adjust-dateTime-to-timezone($srcdt, $timezone)`]
- The result of this function will be a date value that has date and timezone components that are the same as the date and timezone components of `$r`.

10.7.2.1 Examples

Assume the dynamic context provides an implicit timezone of `-05:00 (-PT5H0M)`.

```
let $tz := xs:dayTimeDuration("-PT10H")
```

- `fn:adjust-date-to-timezone(xs:date("2002-03-07"))` returns `2002-03-07-05:00`.
- `fn:adjust-date-to-timezone(xs:date("2002-03-07-07:00"))` returns `2002-03-07-05:00`. `$arg` is converted to the `xs:dateTime "2002-03-07T00:00:00-07:00"`. This is adjusted to the implicit timezone, giving `"2002-03-07T02:00:00-05:00"`.
- `fn:adjust-date-to-timezone(xs:date("2002-03-07"), $tz)` returns `2002-03-07-10:00`.
- `fn:adjust-date-to-timezone(xs:date("2002-03-07-07:00"), $tz)` returns `2002-03-06-10:00`. `$arg` is converted to the `xs:dateTime "2002-03-07T00:00:00-07:00"`. This is adjusted to the given timezone, giving `"2002-03-06T21:00:00-10:00"`.
- `fn:adjust-date-to-timezone(xs:date("2002-03-07"), ())` returns `2002-03-07`.
- `fn:adjust-date-to-timezone(xs:date("2002-03-07-07:00"), ())` returns `2002-03-07`.

10.7.3 fn:adjust-time-to-timezone

```
fn:adjust-time-to-timezone($arg as xs:time?) as xs:time?
```

```
fn:adjust-time-to-timezone($arg      as xs:time?,
                           $timezone as xs:dayTimeDuration?) as xs:time?
```

Summary: Adjusts an `xs:time` value to a specific timezone, or to no timezone at all. If `$timezone` is the empty sequence, returns an `xs:time` without a timezone. Otherwise, returns an `xs:time` with a timezone.

If `$timezone` is not specified, then `$timezone` is the value of the implicit timezone in the dynamic context.

If `$arg` is the empty sequence, then the result is the empty sequence.

A dynamic error is raised [err:FODT0003] if `$timezone` is less than `-PT14H` or greater than `PT14H` or if does not contain an integral number of minutes.

If `$arg` does not have a timezone component and `$timezone` is the empty sequence, then the result is `$arg`.

If `$arg` does not have a timezone component and `$timezone` is not the empty sequence, then the result is `$arg` with `$timezone` as the timezone component.

If `$arg` has a timezone component and `$timezone` is the empty sequence, then the result is the localized value of `$arg` without its timezone component.

If `$arg` has a timezone component and `$timezone` is not the empty sequence, then:

- Let `$srcdt` be an `xs:dateTime` value, with an arbitrary date for the date component and time and timezone components that are the same as the time and timezone components of `$arg`.
- Let `$r` be the result of evaluating

  [`fn:adjust-dateTime-to-timezone($srcdt, $timezone)`]
- The result of this function will be a time value that has time and timezone components that are the same as the time and timezone components of `$r`.

10.7.3.1 Examples

Assume the dynamic context provides an implicit timezone of `-05:00 (-PT5H0M)`.

```
let $tz := xs:dayTimeDuration("-PT10H")
```

- `fn:adjust-time-to-timezone(xs:time("10:00:00"))` returns `10:00:00-05:00`
- `fn:adjust-time-to-timezone(xs:time("10:00:00-07:00"))` returns `12:00:00-05:00`
- `fn:adjust-time-to-timezone(xs:time("10:00:00"), $tz)` returns `10:00:00-10:00`
- `fn:adjust-time-to-timezone(xs:time("10:00:00-07:00"), $tz)` returns `07:00:00-10:00`
- `fn:adjust-time-to-timezone(xs:time("10:00:00"), ())` returns `10:00:00`
- `fn:adjust-time-to-timezone(xs:time("10:00:00-07:00"), ())` returns `10:00:00`
- `fn:adjust-time-to-timezone(xs:time("10:00:00-07:00"), xs:dayTimeDuration("PT10H"))` returns `03:00:00+10:00`

10.8 Arithmetic Operators on Durations, Dates and Times

These functions support adding or subtracting a duration value to or from an `xs:dateTime`, an `xs:date` or an `xs:time` value. Appendix E of [XML Schema Part 2: Datatypes Second Edition] describes an algorithm for performing such operations.

Function	Meaning
`op:subtract-dateTimes`	Returns the difference between two `xs:dateTimes` as an `xs:dayTimeDuration`.
`op:subtract-dates`	Returns the difference between two `xs:dateTimes` as an `xs:dayTimeDuration`.
`op:subtract-times`	Returns the difference between two `xs:times` as an `xs:dayTimeDuration`.
`op:add-yearMonthDuration-to-dateTime`	Returns the end of a time period by adding an `xs:yearMonthDuration` to the `xs:dateTime` that starts the period.
`op:add-dayTimeDuration-to-dateTime`	Returns the end of a time period by adding an `xs:dayTimeDuration` to the `xs:dateTime` that starts the period.
`op:subtract-yearMonthDuration-from-dateTime`	Returns the beginning of a time period by subtracting an `xs:yearMonthDuration` from the `xs:dateTime` that ends the period.
`op:subtract-dayTimeDuration-from-dateTime`	Returns the beginning of a time period by subtracting an `xs:dayTimeDuration` from the `xs:dateTime` that ends the period.

Function	Meaning
op:add-yearMonthDuration-to-date	Returns the end of a time period by adding an `xs:yearMonthDuration` to the `xs:date` that starts the period.
op:add-dayTimeDuration-to-date	Returns the end of a time period by adding an `xs:dayTimeDuration` to the `xs:date` that starts the period.
op:subtract-yearMonthDuration-from-date	Returns the beginning of a time period by subtracting an `xs:yearMonthDuration` from the `xs:date` that ends the period.
op:subtract-dayTimeDuration-from-date	Returns the beginning of a time period by subtracting an `xs:dayTimeDuration` from the `xs:date` that ends the period.
op:add-dayTimeDuration-to-time	Adds the value of the hours, minutes and seconds components of an `xs:dayTimeDuration` to an `xs:time` value.
op:subtract-dayTimeDuration-from-time	Subtracts the value of the hours, minutes and seconds components of an `xs:dayTimeDuration` to an `xs:time` value.

10.8.1 op:subtract-dateTimes

```
op:subtract-dateTimes($arg1 as xs:dateTime,
                      $arg2 as xs:dateTime) as xs:dayTimeDuration
```

Summary: Returns the `xs:dayTimeDuration` that corresponds to the difference between the normalized value of `$arg1` and the normalized value of `$arg2`. If either `$arg1` or `$arg2` do not contain an explicit timezone then, for the purpose of the operation, the implicit timezone provided by the dynamic context (See Section C.2 Dynamic Context Components^XP.) is assumed to be present as part of the value.

If the normalized value of `$arg1` precedes in time the normalized value of `$arg2`, then the returned value is a negative duration.

Backs up the subtract, "-", operator on `xs:dateTime` values.

10.8.1.1 Examples

Assume that the dynamic context provides an implicit timezone value of `-05:00`.

- `op:subtract-dateTimes(xs:dateTime("2000-10-30T06:12:00"), xs:dateTime("1999-11-28T09:00:00Z"))` returns an `xs:dayTimeDuration` value corresponding to 337 days, 2 hours and 12 minutes.

10.8.2 op:subtract-dates

```
op:subtract-dates($arg1 as xs:date, $arg2 as xs:date) as xs:dayTimeDuration
```

Summary: Returns the `xs:dayTimeDuration` that corresponds to the difference between the starting instant of `$arg1` and the the starting instant of `$arg2`. If either `$arg1` or `$arg2` do not contain an explicit timezone then, for the purpose of the operation, the implicit timezone provided by the dynamic context (See Section C.2 Dynamic Context Components^XP.) is assumed to be present as part of the value.

The starting instant of an `xs:date` is the `xs:dateTime` at `00:00:00` on that date.

The result is the result of subtracting the two starting instants using `op:subtract-dateTimes`.

If the starting instant of `$arg1` precedes in time the starting instant of `$arg2`, then the returned value is a negative duration.

Backs up the subtract, "-", operator on `xs:date` values.

10.8.2.1 Examples

- Assume that the dynamic context provides an implicit timezone value of `Z`. `op:subtract-dates(xs:date("2000-10-30"), xs:date("1999-11-28"))` returns an `xs:dayTimeDuration` value corresponding to 337 days. The normalized values of the two starting instants are `{2000, 10, 30, 0, 0, 0, PT0S}` and `{1999, 11, 28, 0, 0, 0, PT0S}`.
- If the dynamic context provides an implicit timezone value of `+05:00`, `op:subtract-dates(xs:date("2000-10-30"), xs:date("1999-11-28Z"))` returns an `xs:dayTimeDuration` value corresponding to 336 days and 19 hours. The normalized values of the two starting instants are `{2000, 10, 29, 19, 0, 0, PT0S}` and `{1999, 11, 28, 0, 0, 0, PT0S}`.
- `op:subtract-dates(xs:date("2000-10-15-05:00"), xs:date("2000-10-10+02:00"))` returns an `xs:dayTimeDuration` value corresponding to lexical form "`P5DT7H`".

10.8.3 op:subtract-times

```
op:subtract-times($arg1 as xs:time, $arg2 as xs:time) as xs:dayTimeDuration
```

Summary: Returns the `xs:dayTimeDuration` that corresponds to the difference between the value of `$arg1` converted to an `xs:dateTime` using the date components from the reference `xs:dateTime` and the value of `$arg2` converted to an `xs:dateTime` using the date components from the same reference `xs:dateTime` . If either `$arg1` or `$arg2` do not contain an explicit timezone then, for the purpose of the operation, the implicit timezone provided by the dynamic context (See Section C.2 Dynamic Context Components^XP.) is assumed to be present as part of the value.

The result is the result of subtracting the two `xs:dateTime`s using `op:subtract-dateTimes`.

If the value of `$arg1` converted to an `xs:dateTime` using the date components from the reference `xs:dateTime` precedes in time the value of `$arg2` converted to an `xs:dateTime` using the date components from the same reference `xs:dateTime` , then the returned value is a negative duration.

Backs up the subtract, "-", operator on `xs:time` values.

10.8.3.1 Examples

Assume that the dynamic context provides an implicit timezone value of `-05:00`. Assume, also, that the date components of the reference `xs:dateTime` correspond to `"1972-12-31"`.

- `op:subtract-times(xs:time("11:12:00Z"), xs:time("04:00:00"))` returns an `xs:dayTimeDuration` value corresponding to 2 hours and 12 minutes. This is obtained by subtracting from the `xs:dateTime` value `{1972, 12, 31, 11, 12, 0, PT0S}` the `xs:dateTime` value `{1972, 12, 31, 9, 0, 0, PT0S}`.
- `op:subtract-times(xs:time("11:00:00-05:00"), xs:time("21:30:00+05:30"))` returns a zero `xs:dayTimeDuration` value corresponding to the lexical representation `"PT0S"`. The two `xs:dateTime` values are `{1972, 12, 31, 11, 0, 0, -PT5H}` and `{1972, 12, 31, 21, 30, 0, PT5H30M}`. These normalize to `{1972, 12, 31, 16, 0, 0, PT0S}` and `{1972, 12, 31, 16, 0, 0, PT0S}`.

- `op:subtract-times(xs:time("17:00:00-06:00"), xs:time("08:00:00+09:00"))` returns an `xs:dayTimeDuration` value corresponding to one day or 24 hours. The two normalized `xs:dateTime` values are `{1972, 12, 31, 23, 0, 0, PT0S}` and `{1972, 12, 30, 23, 0, 0, PT0S}`.
- `op:subtract-times(xs:time("24:00:00"), xs:time("23:59:59"))` returns an `xs:dayTimeDuration` value corresponding to `"-PT23H59M59S"`. The two normalized `xs:dateTime` values are `{1972, 12, 31, 0, 0, 0, ()}` and `{1972, 12, 31, 23, 59, 59.0, ()}`.

10.8.4 op:add-yearMonthDuration-to-dateTime

```
op:add-yearMonthDuration-to-dateTime($arg1 as xs:dateTime,
                                     $arg2 as xs:yearMonthDuration) as xs:dateTime
```

Summary: Returns the `xs:dateTime` computed by adding `$arg2` to the value of `$arg1` using the algorithm described in Appendix E of [XML Schema Part 2: Datatypes Second Edition] disregarding the rule about leap seconds. If `$arg2` is negative, then the result `xs:dateTime` precedes `$arg1`.

The result has the same timezone as `$arg1`. If `$arg1` has no timezone, the result has no timezone.

This functions backs up the "+" operator on `xs:dateTime` and `xs:yearMonthDuration` values.

10.8.4.1 Examples

- `op:add-yearMonthDuration-to-dateTime(xs:dateTime("2000-10-30T11:12:00"), xs:yearMonthDuration("P1Y2M"))` returns an `xs:dateTime` value corresponding to the lexical representation `"2001-12-30T11:12:00"`.

10.8.5 op:add-dayTimeDuration-to-dateTime

```
op:add-dayTimeDuration-to-dateTime($arg1 as xs:dateTime,
                                   $arg2 as xs:dayTimeDuration) as xs:dateTime
```

Summary: Returns the `xs:dateTime` computed by adding `$arg2` to the value of `$arg1` using the algorithm described in Appendix E of [XML Schema Part 2: Datatypes Second Edition] disregarding the rule about leap seconds. If `$arg2` is negative, then the result `xs:dateTime` precedes `$arg1`.

The result has the same timezone as `$arg1`. If `$arg1` has no timezone, the result has no timezone.

This functions backs up the "+" operator on `xs:dateTime` and `xs:dayTimeDuration` values.

10.8.5.1 Examples

- `op:add-dayTimeDuration-to-dateTime(xs:dateTime("2000-10-30T11:12:00"), xs:dayTimeDuration("P3DT1H15M"))` returns an `xs:dateTime` value corresponding to the lexical representation `"2000-11-02T12:27:00"`.

10.8.6 op:subtract-yearMonthDuration-from-dateTime

```
op:subtract-yearMonthDuration-from-dateTime($arg1 as xs:dateTime,
                                            $arg2 as xs:yearMonthDuration) as xs:dateTime
```

Summary: Returns the `xs:dateTime` computed by negating `$arg2` and adding the result to the value of `$arg1` using the algorithm described in Appendix E of [XML Schema Part 2: Datatypes Second Edition] disregarding the rule about leap seconds. If `$arg2` is negative, then the `xs:dateTime` returned follows `$arg1`.

The result has the same timezone as `$arg1`. If `$arg1` has no timezone, the result has no timezone.

This functions backs up the "-" operator on `xs:dateTime` and `xs:yearMonthDuration` values.

10.8.6.1 Examples

- `op:subtract-yearMonthDuration-from-dateTime(xs:dateTime("2000-10-30T11:12:00"), xs:yearMonthDuration("P1Y2M"))` returns an `xs:dateTime` value corresponding to the lexical representation `"1999-08-30T11:12:00"`.

10.8.7 op:subtract-dayTimeDuration-from-dateTime

```
op:subtract-dayTimeDuration-from-dateTime($arg1 as xs:dateTime,
                                          $arg2 as xs:dayTimeDuration) as xs:dateTime
```

Summary: Returns the `xs:dateTime` computed by negating `$arg2` and adding the result to the value of `$arg1` using the algorithm described in Appendix E of [XML Schema Part 2: Datatypes Second Edition] disregarding the rule about leap seconds. If `$arg2` is negative, then the `xs:dateTime` returned follows `$arg1`.

The result has the same timezone as `$arg1`. If `$arg1` has no timezone, the result has no timezone.

This functions backs up the "-" operator on `xs:dateTime` and `xs:dayTimeDuration` values.

10.8.7.1 Examples

- `op:subtract-dayTimeDuration-from-dateTime(xs:dateTime("2000-10-30T11:12:00"), xs:dayTimeDuration("P3DT1H15M"))` returns an `xs:dateTime` value corresponding to the lexical representation `"2000-10-27T09:57:00"`.

10.8.8 op:add-yearMonthDuration-to-date

```
op:add-yearMonthDuration-to-date($arg1 as xs:date,
                                 $arg2 as xs:yearMonthDuration) as xs:date
```

Summary: Returns the `xs:date` computed by adding `$arg2` to the starting instant of `$arg1` using the algorithm described in Appendix E of [XML Schema Part 2: Datatypes Second Edition] and discarding the time components from the resulting `xs:dateTime`. If `$arg2` is negative, then the `xs:date` returned precedes `$arg1`.

The starting instant of an `xs:date` is the `xs:dateTime` at time `00:00:00` on that date.

The result has the same timezone as `$arg1`. If `$arg1` has no timezone, the result has no timezone.

This functions backs up the "+" operator on `xs:date` and `xs:yearMonthDuration` values.

10.8.8.1 Examples

- `op:add-yearMonthDuration-to-date(xs:date("2000-10-30"), xs:yearMonthDuration("P1Y2M"))` returns the `xs:date` corresponding to December 30, 2001.

10.8.9 op:add-dayTimeDuration-to-date

```
op:add-dayTimeDuration-to-date($arg1 as xs:date,
                               $arg2 as xs:dayTimeDuration) as xs:date
```

Summary: Returns the `xs:date` computed by adding `$arg2` to the starting instant of `$arg1` using the algorithm described in Appendix E of [XML Schema Part 2: Datatypes Second Edition] and discarding the time components from the resulting `xs:dateTime`. If `$arg2` is negative, then the `xs:date` returned precedes `$arg1`.

The starting instant of an `xs:date` is the `xs:dateTime` at time `00:00:00` on that date.

The result has the same timezone as `$arg1`. If `$arg1` has no timezone, the result has no timezone.

This functions backs up the "+" operator on `xs:date` and `xs:dayTimeDuration` values.

10.8.9.1 Examples

- `op:add-dayTimeDuration-to-date(xs:date("2004-10-30Z"), xs:dayTimeDuration("P2DT2H30M0S"))` returns the `xs:date` November 1, 2004. The starting instant of the first argument is the `xs:dateTime` value `{2004, 10, 30, 0, 0, 0, PT0S}`. Adding the second argument to this, gives the `xs:dateTime` value `{2004, 11, 1, 2, 30, 0, PT0S}`. The time components are then discarded.

10.8.10 op:subtract-yearMonthDuration-from-date

```
op:subtract-yearMonthDuration-from-date($arg1 as xs:date,
                                        $arg2 as xs:yearMonthDuration) as xs:date
```

Summary: Returns the `xs:date` computed by negating `$arg2` and adding the result to the starting instant of `$arg1` using the algorithm described in Appendix E of [XML Schema Part 2: Datatypes Second Edition] and discarding the time components from the resulting `xs:dateTime`. If `$arg2` is positive, then the `xs:date` returned precedes `$arg1`.

The starting instant of an `xs:date` is the `xs:dateTime` at `00:00:00` on that date.

The result has the same timezone as `$arg1`. If `$arg1` has no timezone, the result has no timezone.

This functions backs up the "-" operator on `xs:date` and `xs:yearMonthDuration` values.

10.8.10.1 Examples

- `op:subtract-yearMonthDuration-from-date(xs:date("2000-10-30"), xs:yearMonthDuration("P1Y2M"))` returns the `xs:date` August 30, 1999.
- `op:subtract-yearMonthDuration-from-date(xs:date("2000-02-29Z"), xs:yearMonthDuration("P1Y"))` returns the `xs:date` February 28, 1999 in timezone `Z`.
- `op:subtract-yearMonthDuration-from-date(xs:date("2000-10-31-05:00"), xs:yearMonthDuration("P1Y1M"))` returns the `xs:date` September 30, 1999 in timezone `-05:00`.

10.8.11 op:subtract-dayTimeDuration-from-date

```
op:subtract-dayTimeDuration-from-date($arg1 as xs:date,
                                      $arg2 as xs:dayTimeDuration) as xs:date
```

Summary: Returns the `xs:date` computed by negating `$arg2` and adding the result to the starting instant of `$arg1` using the algorithm described in Appendix E of [XML Schema Part 2: Datatypes Second Edition] and discarding the time components from the resulting `xs:dateTime`. If `$arg2` is positive, then the `xs:date` returned precedes `$arg1`.

The starting instant of an `xs:date` is the `xs:dateTime` at `00:00:00` on that date.

The result has the same timezone as `$arg1`. If `$arg1` has no timezone, the result has no timezone.

This functions backs up the "-" operator on `xs:date` and `xs:dayTimeDuration` values.

10.8.11.1 Examples

- `op:subtract-dayTimeDuration-from-date(xs:date("2000-10-30"), xs:dayTimeDuration("P3DT1H15M"))` returns the `xs:date` October 26, 2000.

10.8.12 op:add-dayTimeDuration-to-time

```
op:add-dayTimeDuration-to-time($arg1 as xs:time,
                               $arg2 as xs:dayTimeDuration) as xs:time
```

Summary: First, the days component in the canonical lexical representation of `$arg2` is set to zero (0) and the value of the resulting `xs:dayTimeDuration` is calculated. Alternatively, the value of `$arg2` modulus 86,400 is used as the second argument. This value is added to the value of `$arg1` converted to an `xs:dateTime` using a reference date such as `1972-12-31` and the time components of the result returned. Note that the `xs:time` returned may occur in a following or preceding day and may be less than `$arg1`.

The result has the same timezone as `$arg1`. If `$arg1` has no timezone, the result has no timezone.

This functions backs up the "+" operator on `xs:time` and `xs:dayTimeDuration` values.

10.8.12.1 Examples

- `op:add-dayTimeDuration-to-time(xs:time("11:12:00"), xs:dayTimeDuration("P3DT1H15M"))` returns the `xs:time` value corresponding to the lexical representation "`12:27:00`".
- `op:add-dayTimeDuration-to-time(xs:time("23:12:00+03:00"), xs:dayTimeDuration("P1DT3H15M"))` returns the `xs:time` value corresponding to the lexical representation "`02:27:00+03:00`", i.e. `{0, 0, 0, 2, 27, 0, PT3H}`.

10.8.13 op:subtract-dayTimeDuration-from-time

```
op:subtract-dayTimeDuration-from-time($arg1 as xs:time,
                                      $arg2 as xs:dayTimeDuration) as xs:time
```

Summary: The result is calculated by first setting the day component in the canonical lexical representation of `$arg2` to zero (0) and calculating the value of the resulting `xs:dayTimeDuration`. Alternatively, the value of `$arg2` modulus 86,400 is used as the second argument. This value is subtracted from the value of `$arg1` converted to an `xs:dateTime` using a reference date such as

`1972-12-31` and the time components of the result are returned. Note that the `xs:time` returned may occur in a preceding or following day and may be greater than `$arg1`.

The result has the same timezone as `$arg1`. If `$arg1` has no timezone, the result has no timezone.

This functions backs up the "-" operator on `xs:time` and `xs:dayTimeDuration` values.

10.8.13.1 Examples

- `op:subtract-dayTimeDuration-from-time(xs:time("11:12:00"), xs:dayTimeDuration("P3DT1H15M"))` returns an `xs:time` value corresponding to the lexical representation `"09:57:00"`.
- `op:subtract-dayTimeDuration-from-time(xs:time("08:20:00-05:00"), xs:dayTimeDuration("P23DT10H10M"))` returns the `xs:time` value corresponding to the lexical representation `"22:10:00-05:00"` i.e. `{0, 0, 0, 22, 10, 0, -PT5H}`

11 Functions Related to QNames

11.1 Additional Constructor Functions for QNames

This section defines additional constructor functions for QName as defined in [XML Schema Part 2: Datatypes Second Edition]. Leading and trailing whitespace, if present, is stripped from string arguments before the result is constructed.

Function	Meaning
`fn:resolve-QName`	Returns an `xs:QName` with the lexical form given in the first argument. The prefix is resolved using the in-scope namespaces for a given element.
`fn:QName`	Returns an `xs:QName` with the namespace URI given in the first argument and the local name and prefix in the second argument.

11.1.1 fn:resolve-QName

```
fn:resolve-QName($qname as xs:string?, $element as element()) as xs:QName?
```

Summary: Returns an `xs:QName` value (that is, an expanded-QName) by taking an `xs:string` that has the lexical form of an `xs:QName` (a string in the form "prefix:local-name" or "local-name") and resolving it using the in-scope namespaces for a given element.

If `$qname` does not have the correct lexical form for `xs:QName` an error is raised [err:FOCA0002].

If `$qname` is the empty sequence, returns the empty sequence.

More specifically, the function searches the namespace bindings of `$element` for a binding whose name matches the prefix of `$qname`, or the zero-length string if it has no prefix, and constructs an expanded-QName whose local name is taken from the supplied `$qname`, and whose namespace URI is taken from the string value of the namespace binding.

If the `$qname` has a prefix and if there is no namespace binding for `$element` that matches this prefix, then an error is raised [err:FONS0004].

If the `$qname` has no prefix, and there is no namespace binding for `$element` corresponding to the default (unnamed) namespace, then the resulting expanded-QName has no namespace part.

The prefix (or absence of a prefix) in the supplied `$qname` argument is retained in the returned expanded-QName, as discussed in [Section 2.1 Terminology](DM)^{DM}.

11.1.1.1 Usage Note

Sometimes the requirement is to construct an `xs:QName` without using the default namespace. This can be achieved by writing:

```
if (contains($qname, ":")) then fn:resolve-QName($qname, $element) else
                             fn:QName("", $qname)
```

If the requirement is to construct an `xs:QName` using the namespaces in the static context, then the `xs:QName` constructor should be used.

11.1.1.2 Examples

Assume that the element bound to `$element` has a single namespace binding bound to the prefix `eg`.

- `fn:resolve-QName("hello", $element)` returns a QName with local name "hello" that is in no namespace.
- `fn:resolve-QName("eg:myFunc", $element)` returns an `xs:QName` whose namespace URI is specified by the namespace binding corresponding to the prefix "eg" and whose local name is "myFunc".

11.1.2 fn:QName

```
fn:QName($paramURI as xs:string?, $paramQName as xs:string) as xs:QName
```

Summary: Returns an `xs:QName` with the namespace URI given in `$paramURI`. If `$paramURI` is the zero-length string or the empty sequence, it represents "no namespace"; in this case, if the value of `$paramQName` contains a colon (`:`), an error is raised [err:FOCA0002]. The prefix (or absence of a prefix) in `$paramQName` is retained in the returned `xs:QName` value. The local name in the result is taken from the local part of `$paramQName`.

If `$paramQName` does not have the correct lexical form for `xs:QName` an error is raised [err:FOCA0002].

Note that unlike `xs:QName` this function does not require a `xs:string` literal as the argument.

11.1.2.1 Examples

- `fn:QName("http://www.example.com/example", "person")` returns an `xs:QName` with namespace URI = "http://www.example.com/example", local name = "person" and prefix = "".
- `fn:QName("http://www.example.com/example", "ht:person")` returns an `xs:QName` with namespace URI = "http://www.example.com/example", local name = "person" and prefix = "ht".

11.2 Functions and Operators Related to QNames

This section discusses functions on QNames as defined in [XML Schema Part 2: Datatypes Second Edition].

Function	Meaning

Function	Meaning
op:QName-equal	Returns `true` if the local names and namespace URIs of the two arguments are equal.
fn:prefix-from-QName	Returns an `xs:NCName` representing the prefix of the `xs:QName` argument.
fn:local-name-from-QName	Returns an `xs:NCName` representing the local name of the `xs:QName` argument.
fn:namespace-uri-from-QName	Returns the namespace URI for the `xs:QName` argument. If the `xs:QName` is in no namespace, the zero-length string is returned.
fn:namespace-uri-for-prefix	Returns the namespace URI of one of the in-scope namespaces for the given element, identified by its namespace prefix.
fn:in-scope-prefixes	Returns the prefixes of the in-scope namespaces for the given element.

11.2.1 op:QName-equal

```
op:QName-equal($arg1 as xs:QName, $arg2 as xs:QName) as xs:boolean
```

Summary: Returns `true` if the namespace URIs of `$arg1` and `$arg2` are equal and the local names of `$arg1` and `$arg2` are identical based on the Unicode code point collation (`http://www.w3.org/2005/xpath-functions/collation/codepoint`). Otherwise, returns `false`. Two namespace URIs are considered equal if they are either both absent or both present and identical based on the Unicode code point collation. The prefix parts of `$arg1` and `$arg2`, if any, are ignored.

Backs up the "eq" and "ne" operators on values of type `xs:QName`.

11.2.2 fn:prefix-from-QName

```
fn:prefix-from-QName($arg as xs:QName?) as xs:NCName?
```

Summary: Returns an `xs:NCName` representing the prefix of `$arg`. The empty sequence is returned if `$arg` is the empty sequence or if the value of `$arg` contains no prefix.

11.2.3 fn:local-name-from-QName

```
fn:local-name-from-QName($arg as xs:QName?) as xs:NCName?
```

Summary: Returns an `xs:NCName` representing the local part of `$arg`. If `$arg` is the empty sequence, returns the empty sequence.

11.2.3.1 Examples

- `fn:local-name-from-QName(fn:QName("http://www.example.com/example", "person"))` returns `"person"`.

11.2.4 fn:namespace-uri-from-QName

```
fn:namespace-uri-from-QName($arg as xs:QName?) as xs:anyURI?
```

Summary: Returns the namespace URI for `$arg` as an `xs:anyURI`. If `$arg` is the empty sequence, the empty sequence is returned. If `$arg` is in no namespace, the zero-length `xs:anyURI` is returned.

11.2.4.1 Examples

- `fn:namespace-uri-from-QName(fn:QName("http://www.example.com/example", "person"))` returns the namespace URI corresponding to `"http://www.example.com/example"`.

11.2.5 fn:namespace-uri-for-prefix

```
fn:namespace-uri-for-prefix($prefix  as xs:string?,
                            $element as element()) as xs:anyURI?
```

Summary: Returns the namespace URI of one of the in-scope namespaces for `$element`, identified by its namespace prefix.

If `$element` has an in-scope namespace whose namespace prefix is equal to `$prefix`, it returns the namespace URI of that namespace. If `$prefix` is the zero-length string or the empty sequence, it returns the namespace URI of the default (unnamed) namespace. Otherwise, it returns the empty sequence.

Prefixes are equal only if their Unicode code points match exactly.

11.2.6 fn:in-scope-prefixes

```
fn:in-scope-prefixes($element as element()) as xs:string*
```

Summary: Returns the prefixes of the in-scope namespaces for `$element`. For namespaces that have a prefix, it returns the prefix as an `xs:NCName`. For the default namespace, which has no prefix, it returns the zero-length string.

12 Operators on base64Binary and hexBinary

12.1 Comparisons of base64Binary and hexBinary Values

The following comparison operators on `xs:base64Binary` and `xs:hexBinary` values are defined. Comparisons take two operands of the same type; that is, both operands must be `xs:base64Binary` or both operands may be `xs:hexBinary`. Each returns a boolean value.

A value of type `xs:hexBinary` can be compared with a value of type `xs:base64Binary` by casting one value to the other type. See **17.1.7 Casting to xs:base64Binary and xs:hexBinary**.

Function	Meaning
`op:hexBinary-equal`	Returns `true` if the two arguments are equal.
`op:base64Binary-equal`	Returns `true` if the two arguments are equal.

12.1.1 op:hexBinary-equal

```
op:hexBinary-equal($value1 as xs:hexBinary,
                   $value2 as xs:hexBinary) as xs:boolean
```

Summary: Returns `true` if `$value1` and `$value2` are of the same length, measured in binary octets, and contain the same octets in the same order. Otherwise, returns `false`.

This function backs up the "eq" and "ne" operators on `xs:hexBinary` values.

12.1.2 op:base64Binary-equal

```
op:base64Binary-equal($value1 as xs:base64Binary,
                      $value2 as xs:base64Binary) as xs:boolean
```

Summary: Returns `true` if `$value1` and `$value2` are of the same length, measured in binary octets, and contain the same octets in the same order. Otherwise, returns `false`.

This function backs up the "eq" and "ne" operators on `xs:base64Binary` values.

13 Operators on NOTATION

13.1 Operators on NOTATION

This section discusses functions that take NOTATION as arguments.

Function	Meaning
`op:NOTATION-equal`	Returns `true` if the two arguments are `op:QName-equal`.

13.1.1 op:NOTATION-equal

```
op:NOTATION-equal($arg1 as xs:NOTATION, $arg2 as xs:NOTATION) as xs:boolean
```

Summary: Returns `true` if the namespace URIs of `$arg1` and `$arg2` are equal and the local names of `$arg1` and `$arg2` are identical based on the Unicode code point collation: `http://www.w3.org/2005/xpath-functions/collation/codepoint`. Otherwise, returns false. Two namespace URIs are considered equal if they are either both absent or both present and identical based on the Unicode code point collation. The prefix parts of `$arg1` and `$arg2`, if any, are ignored.

Backs up the "eq" and "ne" operators on values of type `xs:NOTATION`.

14 Functions and Operators on Nodes

This section discusses functions and operators on nodes. Nodes are formally defined in Section 6 Nodes^DM.

Function	Meaning
`fn:name`	Returns the name of the context node or the specified node as an `xs:string`.
`fn:local-name`	Returns the local name of the context node or the specified node as an `xs:NCName`.
`fn:namespace-uri`	Returns the namespace URI as an `xs:anyURI` for the `xs:QName` of the argument node or the context node if the argument is omitted. This may be the URI corresponding to the zero-length string if the `xs:QName` is in no namespace.
`fn:number`	Returns the value of the context item after atomization or the specified argument converted to an `xs:double`.

Function	Meaning
fn:lang	Returns `true` or `false`, depending on whether the language of the given node or the context node, as defined using the xml:lang attribute, is the same as, or a sublanguage of, the language specified by the argument.
op:is-same-node	Returns `true` if the two arguments have the same identity.
op:node-before	Indicates whether one node appears before another node in document order.
op:node-after	Indicates whether one node appears after another node in document order.
fn:root	Returns the root of the tree to which the node argument belongs.

For the illustrative examples below assume an XQuery or transformation operating on a PurchaseOrder document containing a number of line-item elements. Each line-item has child elements called description, price, quantity, etc. whose content is different for each line-item. Quantity has simple content of type `xs:decimal`. Further assume that variables `$item1`, `$item2`, etc. are each bound to single line-item element nodes in the document in sequence and that the value of the quantity child of the first line-item is `5.0`.

```
<PurchaseOrder>
  <line-item>
    <description> ... </description>
    <price> ... </price>
    <quantity>5.0</quantity>
      ...
  </line-item>
  <line-item>
      ...
  </line-item>
      ...
</PurchaseOrder>
```

14.1 fn:name

```
fn:name() as xs:string
fn:name($arg as node()?) as xs:string
```

Summary: Returns the name of a node, as an `xs:string` that is either the zero-length string, or has the lexical form of an `xs:QName`.

If the argument is omitted, it defaults to the context item (`.`). The behavior of the function if the argument is omitted is exactly the same as if the context item had been passed as the argument.

The following errors may be raised: if the context item is undefined [err:XPDY0002]XP; if the context item is not a node [err:XPTY0004]XP.

If the argument is supplied and is the empty sequence, the function returns the zero-length string.

If the target node has no name (that is, if it is a document node, a comment, a text node, or a namespace binding having no name), the function returns the zero-length string.

Otherwise, the value returned is `fn:string(fn:node-name($arg))`.

14.2 fn:local-name

```
fn:local-name() as xs:string
fn:local-name($arg as node()?) as xs:string
```

Summary: Returns the local part of the name of `$arg` as an `xs:string` that will either be the zero-length string or will have the lexical form of an `xs:NCName`.

If the argument is omitted, it defaults to the context item (`.`). The behavior of the function if the argument is omitted is exactly the same as if the context item had been passed as the argument.

The following errors may be raised: if the context item is undefined [err:XPDY0002]XP; if the context item is not a node [err:XPTY0004]XP.

If the argument is supplied and is the empty sequence, the function returns the zero-length string.

If the target node has no name (that is, if it is a document node, a comment, or a text node), the function returns the zero-length string.

Otherwise, the value returned will be the local part of the expanded-QName of the target node (as determined by the `dm:node-name` accessor in Section 5.11 node-name AccessorDM). This will be an `xs:string` whose lexical form is an `xs:NCName`.

14.3 fn:namespace-uri

```
fn:namespace-uri() as xs:anyURI
fn:namespace-uri($arg as node()?) as xs:anyURI
```

Summary: Returns the namespace URI part of the name of `$arg`, as an `xs:anyURI` value.

If the argument is omitted, it defaults to the context node (`.`). The behavior of the function if the argument is omitted is exactly the same as if the context item had been passed as the argument.

The following errors may be raised: if the context item is undefined [err:XPDY0002]XP; if the context item is not a node [err:XPTY0004]XP.

If `$arg` is neither an element nor an attribute node, or if it is an element or attribute node whose expanded-QName (as determined by the `dm:node-name` accessor in the Section 5.11 node-name AccessorDM) is in no namespace, then the function returns the `xs:anyURI` corresponding to the zero-length string.

14.4 fn:number

```
fn:number() as xs:double
fn:number($arg as xs:anyAtomicType?) as xs:double
```

Summary: Returns the value indicated by `$arg` or, if `$arg` is not specified, the context item after atomization, converted to an `xs:double`

Calling the zero-argument version of the function is defined to give the same result as calling the single-argument version with the context item (`.`). That is, `fn:number()` is equivalent to `fn:number(.)`.

If `$arg` is the empty sequence or if `$arg` or the context item cannot be converted to an `xs:double`, the `xs:double` value `NaN` is returned. If the context item is undefined an error is raised: [err:XPDY0002]XP.

If `$arg` is the empty sequence, `NaN` is returned. Otherwise, `$arg`, or the context item after atomization, is converted to an `xs:double` following the rules of **[17.1.3.2 Casting to xs:double](#)**. If the conversion to `xs:double` fails, the `xs:double` value `NaN` is returned.

14.4.1 Examples

- `fn:number($item1/quantity)` returns `5.0`.
- `fn:number($item2/description)` returns `NaN` (assuming the `description` is non-numeric).
- Assume that the context item is the `xs:string` "`15`". `fn:number()` returns `1.5E1`.

14.5 fn:lang

```
fn:lang($testlang as xs:string?) as xs:boolean
```

```
fn:lang($testlang as xs:string?, $node as node()) as xs:boolean
```

Summary: This function tests whether the language of `$node`, or the context item if the second argument is omitted, as specified by `xml:lang` attributes is the same as, or is a sublanguage of, the language specified by `$testlang`. The behavior of the function if the second argument is omitted is exactly the same as if the context item (`.`) had been passed as the second argument. The language of the argument node, or the context item if the second argument is omitted, is determined by the value of the `xml:lang` attribute on the node, or, if the node has no such attribute, by the value of the `xml:lang` attribute on the nearest ancestor of the node that has an `xml:lang` attribute. If there is no such ancestor, then the function returns `false`

The following errors may be raised: if the context item is undefined [err:XPDY0002]*XP*; if the context item is not a node [err:XPTY0004]*XP*.

If `$testlang` is the empty sequence it is interpreted as the zero-length string.

The relevant `xml:lang` attribute is determined by the value of the XPath expression:

```
(ancestor-or-self::*/@xml:lang)[last()]
```

If this expression returns an empty sequence, the function returns `false`.

Otherwise, the function returns `true` if and only if, based on a caseless default match as specified in section 3.13 of [The Unicode Standard], either:

1. `$testlang` is equal to the string-value of the relevant `xml:lang` attribute, or
2. `$testlang` is equal to some substring of the string-value of the relevant `xml:lang` attribute that starts at the start of the string-value and ends immediately before a hyphen, "-" (The character "-" is HYPHEN-MINUS, #x002D).

14.5.1 Examples

- The expression `fn:lang("en")` would return `true` if the context node were any of the following four elements:
    - `<para xml:lang="en"/>`
    - `<div xml:lang="en"><para>And now, and forever!</para></div>`
    - `<para xml:lang="EN"/>`
    - `<para xml:lang="en-us"/>`

- The expression `fn:lang("fr")` would return `false` if the context node were `<para xml:lang="EN"/>`

14.6 op:is-same-node

```
op:is-same-node($parameter1 as node(), $parameter2 as node()) as xs:boolean
```

Summary: If the node identified by the value of `$parameter1` is the same node as the node identified by the value of `$parameter2` (that is, the two nodes have the same identity), then the function returns `true`; otherwise, the function returns `false`. This function backs up the "is" operator on nodes.

14.6.1 Examples

- `op:is-same-node($item1, $item1)` returns `true`.
- `op:is-same-node($item1, $item2)` returns `false`.

14.7 op:node-before

```
op:node-before($parameter1 as node(), $parameter2 as node()) as xs:boolean
```

Summary: If the node identified by the value of `$parameter1` occurs in document order before the node identified by the value of `$parameter2`, this function returns `true`; otherwise, it returns `false`. The rules determining the order of nodes within a single document and in different documents can be found in Section 2.4 Document Order*DM*. This function backs up the "<<" operator.

14.7.1 Examples

- `op:node-before($item1, $item2)` returns `true`.
- `op:node-before($item1, $item1)` returns `false`.

14.8 op:node-after

```
op:node-after($parameter1 as node(), $parameter2 as node()) as xs:boolean
```

Summary: If the node identified by the value of `$parameter1` occurs in document order after the node identified by the value of `$parameter2`, this function returns `true`; otherwise, it returns `false`. The rules determining the order of nodes within a single document and in different documents can be found in Section 2.4 Document Order*DM*. This function backs up the ">>" operator.

14.8.1 Examples

- `op:node-after($item1, $item2)` returns `false`.
- `op:node-after($item1, $item1)` returns `false`.
- `op:node-after($item2, $item1)` returns `true`.

14.9 fn:root

```
fn:root() as node()
```

```
fn:root($arg as node()?) as node()?
```

Summary: Returns the root of the tree to which `$arg` belongs. This will usually, but not necessarily, be a document node.

If `$arg` is the empty sequence, the empty sequence is returned.

If `$arg` is a document node, `$arg` is returned.

If the function is called without an argument, the context item (`.`) is used as the default argument. The behavior of the function if the argument is omitted is exactly the same as if the context item had been passed as the argument.

The following errors may be raised: if the context item is undefined [err:XPDY0002]*XP*; if the context item is not a node [err:XPTY0004]*XP*.

14.9.1 Examples

These examples use some variables which could be defined in [XQuery 1.0: An XML Query Language] as:

```
let $i := <tool>wrench</tool>
let $o := <order> {$i} <quantity>5</quantity> </order>
let $odoc := document {$o}
let $newi := $o/tool
```

Or they could be defined in [XSL Transformations (XSLT) Version 2.0] as:

```
<xsl:variable name="i" as="element()">
  <tool>wrench</tool>
</xsl:variable>

<xsl:variable name="o" as="element()">
  <order>
    <xsl:copy-of select="$i"/>
    <quantity>5</quantity>
  </order>
</xsl:variable>

<xsl:variable name="odoc">
  <xsl:copy-of select="$o"/>
</xsl:variable>

<xsl:variable name="newi" select="$o/tool"/>
```

- `fn:root($i)` returns `$i`
- `fn:root($o/quantity)` returns `$o`
- `fn:root($odoc//quantity)` returns `$odoc`
- `fn:root($newi)` returns `$o`

  Note:

  The final three examples could be made type-safe by wrapping their operands with fn:exactly-one().

15 Functions and Operators on Sequences

A `sequence` is an ordered collection of zero or more `items`. An `item` is either a node or an atomic value. The terms `sequence` and `item` are defined formally in [XQuery 1.0: An XML Query Language] and [XML Path Language (XPath) 2.0].

15.1 General Functions and Operators on Sequences

The following functions are defined on sequences.

Function	Meaning
`fn:boolean`	Computes the effective boolean value of the argument sequence.
`op:concatenate`	Concatenates two sequences.
`fn:index-of`	Returns a sequence of `xs:integer`s, each of which is the index of a member of the sequence specified as the first argument that is equal to the value of the second argument. If no members of the specified sequence are equal to the value of the second argument, the empty sequence is returned.
`fn:empty`	Indicates whether or not the provided sequence is empty.
`fn:exists`	Indicates whether or not the provided sequence is not empty.
`fn:distinct-values`	Returns a sequence in which all but one of a set of duplicate values, based on value equality, have been deleted. The order in which the distinct values are returned is ·implementation dependent·.
`fn:insert-before`	Inserts an item or sequence of items at a specified position in a sequence.
`fn:remove`	Removes an item from a specified position in a sequence.
`fn:reverse`	Reverses the order of items in a sequence.
`fn:subsequence`	Returns the subsequence of a given sequence, identified by location.
`fn:unordered`	Returns the items in the given sequence in a non-deterministic order.

As in the previous section, for the illustrative examples below, assume an XQuery or transformation operating on a non-empty Purchase Order document containing a number of line-item elements. The variable `$seq` is bound to the sequence of line-item nodes in document order. The variables `$item1`, `$item2`, etc. are bound to separate, individual line-item nodes in the sequence.

15.1.1 fn:boolean

```
fn:boolean($arg as item()*) as xs:boolean
```

Summary: Computes the effective boolean value of the sequence `$arg`. See Section 2.4.3 Effective Boolean Value^XP

- If `$arg` is the empty sequence, `fn:boolean` returns `false`.
- If `$arg` is a sequence whose first item is a node, `fn:boolean` returns `true`.
- If `$arg` is a singleton value of type `xs:boolean` or a derived from `xs:boolean`, `fn:boolean` returns `$arg`.
- If `$arg` is a singleton value of type `xs:string` or a type derived from `xs:string`, `xs:anyURI` or a type derived from `xs:anyURI` or `xs:untypedAtomic`, `fn:boolean` returns `false` if the operand value has zero length; otherwise it returns `true`.
- If `$arg` is a singleton value of any numeric type or a type derived from a numeric type, `fn:boolean` returns `false` if the operand value is `NaN` or is numerically equal to zero; otherwise it returns `true`.

- In all other cases, `fn:boolean` raises a type error [err:FORG0006].

The static semantics of this function are described in Section 7.2.4 The fn:boolean and fn:not functions^FS.

> **Note:**
>
> The result of this function is not necessarily the same as " `$arg cast as xs:boolean` ". For example, `fn:boolean("false")` returns the value `"true"` whereas `"false" cast as xs:boolean` returns `false`.

15.1.1.1 Examples

let `$x := ("a", "b", "c")`

- `fn:boolean($x)` raises a type error [err:FORG0006].
- `fn:boolean($x[1])` returns `true`.
- `fn:boolean($x[0])` returns `false`.

15.1.2 op:concatenate

```
op:concatenate($seq1 as item()*, $seq2 as item()*) as item()*
```

Summary: Returns a sequence consisting of the items in `$seq1` followed by the items in `$seq2`. This function backs up the infix operator ",". If either sequence is the empty sequence, the other operand is returned.

For detailed type semantics, see Section 4.3.1 Constructing Sequences^FS

15.1.2.1 Examples

- `op:concatenate((1, 2, 3), (4, 5))` returns `(1, 2, 3, 4, 5)`.
- `op:concatenate((1, 2, 3), ())` returns `(1, 2, 3)`.
- `op:concatenate((), ())` returns `()`.

15.1.3 fn:index-of

```
fn:index-of($seqParam  as xs:anyAtomicType*,
            $srchParam as xs:anyAtomicType) as xs:integer*
fn:index-of($seqParam  as xs:anyAtomicType*,
            $srchParam as xs:anyAtomicType,
            $collation as xs:string) as xs:integer*
```

Summary: Returns a sequence of positive integers giving the positions within the sequence `$seqParam` of items that are equal to `$srchParam`.

The collation used by the invocation of this function is determined according to the rules in **7.3.1 Collations**. The collation is used when string comparison is required.

The items in the sequence `$seqParam` are compared with `$srchParam` under the rules for the `eq` operator. Values of type `xs:untypedAtomic` are compared as if they were of type `xs:string`. Values that cannot be compared, i.e. the `eq` operator is not defined for their types, are considered to be distinct. If an item compares equal, then the position of that item in the sequence `$seqParam` is included in the result.

If the value of `$seqParam` is the empty sequence, or if no item in `$seqParam` matches `$srchParam`, then the empty sequence is returned.

The first item in a sequence is at position 1, not position 0.

The result sequence is in ascending numeric order.

15.1.3.1 Examples

- `fn:index-of ((10, 20, 30, 40), 35)` returns `()`.
- `fn:index-of ((10, 20, 30, 30, 20, 10), 20)` returns `(2, 5)`.
- `fn:index-of (("a", "sport", "and", "a", "pastime"), "a")` returns `(1, 4)`.
- If `@a` is an attribute of type `xs:NMTOKENS` whose string value is `"red green blue"`, and whose typed value is therefore the sequence of three `xs:NMTOKEN` values `("red", "green", "blue")`, then `fn:index-of(@a, "blue")` returns `3`.

  This is because the function calling mechanism atomizes the attribute node to produce a sequence of three `xs:NMTOKEN`s.

15.1.4 fn:empty

```
fn:empty($arg as item()*) as xs:boolean
```

Summary: If the value of `$arg` is the empty sequence, the function returns `true`; otherwise, the function returns `false`.

15.1.4.1 Examples

- `fn:empty(fn:remove(("hello", "world"), 1))` returns `false`.

15.1.5 fn:exists

```
fn:exists($arg as item()*) as xs:boolean
```

Summary: If the value of `$arg` is not the empty sequence, the function returns `true`; otherwise, the function returns `false`.

15.1.5.1 Examples

- `fn:exists(fn:remove(("hello"), 1))` returns `false`.

15.1.6 fn:distinct-values

```
fn:distinct-values($arg as xs:anyAtomicType*) as xs:anyAtomicType*
fn:distinct-values($arg       as xs:anyAtomicType*,
                   $collation as xs:string) as xs:anyAtomicType*
```

Summary: Returns the sequence that results from removing from `$arg` all but one of a set of values that are `eq` to one other. Values of type `xs:untypedAtomic` are compared as if they were of type `xs:string`. Values that cannot be compared, i.e. the `eq` operator is not defined for their types, are considered to be distinct. The order in which the sequence of values is returned is ·implementation dependent·.

The static type of the result is a sequence of prime types as defined in Section 7.2.7 The fn:distinct-values function^FS.

The collation used by the invocation of this function is determined according to the rules in **7.3.1 Collations**. The collation is used when string comparison is required.

If `$arg` is the empty sequence, the empty sequence is returned.

For `xs:float` and `xs:double` values, positive zero is equal to negative zero and, although `NaN` does not equal itself, if `$arg` contains multiple `NaN` values a single `NaN` is returned.

If `xs:dateTime`, `xs:date` or `xs:time` values do not have a timezone, they are considered to have the implicit timezone provided by the dynamic context for the purpose of comparison. Note that `xs:dateTime`, `xs:date` or `xs:time` values can compare equal even if their timezones are different.

Which value of a set of values that compare equal is returned is ·implementation dependent·.

If the input sequence 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. In the situation where the input contains three values *A*, *B*, and *C* such that `A eq B`, `B eq C`, but `A ne C`, then the number of items in the result of the function (as well as the choice of which items are returned) is ·implementation-dependent·, subject only to the constraints that (a) no two items in the result sequence compare equal to each other, and (b) every input item that does not appear in the result sequence compares equal to some item that does appear in the result sequence.

For example, this arises when computing:

```
        distinct-values(
           (xs:float('1.0'),
            xs:decimal('1.0000000000100000000001',
            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.

15.1.6.1 Examples

- `fn:distinct-values((1, 2.0, 3, 2))` might return `(1, 3, 2.0)`.
- The following query:

  ```
  let $x as xs:untypedAtomic*
      := (xs:untypedAtomic("cherry"),
          xs:untypedAtomic("bar"),
          xs:untypedAtomic("bar"))
  return fn:distinct-values ($x)
  ```

  returns a sequence containing two items of type `xs:untypedAtomic`.

15.1.7 fn:insert-before

```
fn:insert-before($target   as item()*,
                 $position as xs:integer,
                 $inserts  as item()*) as item()*
```

Summary: Returns a new sequence constructed from the value of `$target` with the value of `$inserts` inserted at the position specified by the value of `$position`. (The value of `$target` is not affected by the sequence construction.)

If `$target` is the empty sequence, `$inserts` is returned. If `$inserts` is the empty sequence, `$target` is returned.

The value returned by the function consists of all items of `$target` whose index is less than `$position`, followed by all items of `$inserts`, followed by the remaining elements of `$target`, in that sequence.

If `$position` is less than one (1), the first position, the effective value of `$position` is one (1). If `$position` is greater than the number of items in `$target`, then the effective value of `$position` is equal to the number of items in `$target` plus 1.

For detailed semantics see, Section 7.2.15 The fn:insert-before function[FS].

15.1.7.1 Examples

let `$x` := `("a", "b", "c")`

- `fn:insert-before($x, 0, "z")` returns `("z", "a", "b", "c")`
- `fn:insert-before($x, 1, "z")` returns `("z", "a", "b", "c")`
- `fn:insert-before($x, 2, "z")` returns `("a", "z", "b", "c")`
- `fn:insert-before($x, 3, "z")` returns `("a", "b", "z", "c")`
- `fn:insert-before($x, 4, "z")` returns `("a", "b", "c", "z")`

15.1.8 fn:remove

```
fn:remove($target as item()*, $position as xs:integer) as item()*
```

Summary: Returns a new sequence constructed from the value of `$target` with the item at the position specified by the value of `$position` removed.

If `$position` is less than 1 or greater than the number of items in `$target`, `$target` is returned. Otherwise, the value returned by the function consists of all items of `$target` whose index is less than `$position`, followed by all items of `$target` whose index is greater than `$position`. If `$target` is the empty sequence, the empty sequence is returned.

For detailed type semantics, see Section 7.2.11 The fn:remove function[FS]

15.1.8.1 Examples

let `$x` := `("a", "b", "c")`

- `fn:remove($x, 0)` returns `("a", "b", "c")`
- `fn:remove($x, 1)` returns `("b", "c")`
- `fn:remove($x, 6)` returns `("a", "b", "c")`

- `fn:remove((), 3)` returns `()`

15.1.9 fn:reverse

```
fn:reverse($arg as item()*) as item()*
```

Summary: Reverses the order of items in a sequence. If `$arg` is the empty sequence, the empty sequence is returned.

For detailed type semantics, see Section 7.2.12 The fn:reverse function^FS

15.1.9.1 Examples

let `$x` := `("a", "b", "c")`

- `fn:reverse($x)` returns `("c", "b", "a")`
- `fn:reverse(("hello"))` returns `("hello")`
- `fn:reverse(())` returns `()`

15.1.10 fn:subsequence

```
fn:subsequence($sourceSeq as item()*, $startingLoc as xs:double) as item()*
fn:subsequence($sourceSeq   as item()*,
               $startingLoc as xs:double,
               $length      as xs:double) as item()*
```

Summary: Returns the contiguous sequence of items in the value of `$sourceSeq` beginning at the position indicated by the value of `$startingLoc` and continuing for the number of items indicated by the value of `$length`.

In the two-argument case, returns:

```
$sourceSeq[fn:round($startingLoc) le position()]
```

In the three-argument case, returns:

```
$sourceSeq[fn:round($startingLoc) le position()
    and position() lt fn:round($startingLoc) + fn:round($length)]
```

Notes:

If `$sourceSeq` is the empty sequence, the empty sequence is returned.

If `$startingLoc` is zero or negative, the subsequence includes items from the beginning of the `$sourceSeq`.

If `$length` is not specified, the subsequence includes items to the end of `$sourceSeq`.

If `$length` is greater than the number of items in the value of `$sourceSeq` following `$startingLoc`, the subsequence includes items to the end of `$sourceSeq`.

The first item of a sequence is located at position 1, not position 0.

For detailed type semantics, see Section 7.2.13 The fn:subsequence function[FS].

The reason the function accepts arguments of type `xs:double` is that many computations on untyped data return an `xs:double` result; and the reason for the rounding rules is to compensate for any imprecision in these floating-point computations.

15.1.10.1 Examples

Assume `$seq = ($item1, $item2, $item3, $item4, ...)`

- `fn:subsequence($seq, 4)` returns `($item4, ...)`
- `fn:subsequence($seq, 3, 2)` returns `($item3, $item4)`

15.1.11 fn:unordered

```
fn:unordered($sourceSeq as item()*) as item()*
```

Summary: Returns the items of `$sourceSeq` in an ·implementation dependent· order.

Note:

Query optimizers may be able to do a better job if the order of the output sequence is not specified. For example, when retrieving prices from a purchase order, if an index exists on prices, it may be more efficient to return the prices in index order rather than in document order.

15.2 Functions That Test the Cardinality of Sequences

The following functions test the cardinality of their sequence arguments.

Function	Meaning
`fn:zero-or-one`	Returns the input sequence if it contains zero or one items. Raises an error otherwise.
`fn:one-or-more`	Returns the input sequence if it contains one or more items. Raises an error otherwise.
`fn:exactly-one`	Returns the input sequence if it contains exactly one item. Raises an error otherwise.

The functions `fn:zero-or-one`, `fn:one-or-more`, and `fn:exactly-one` defined in this section, check that the cardinality of a sequence is in the expected range. They are particularly useful with regard to static typing. For example, the XML Schema [XML Schema Part 1: Structures Second Edition] describing the output of a query may require a sequence of length one-or-more in some position, but the static type system may not be able to infer this; inserting a call to `fn:one-or-more` at the appropriate place will provide a suitable static type at query analysis time, and confirm that the length is correct with a dynamic check at query execution time.

15.2.1 fn:zero-or-one

```
fn:zero-or-one($arg as item()*) as item()?
```

Summary: Returns `$arg` if it contains zero or one items. Otherwise, raises an error [err:FORG0003].

For detailed type semantics, see Section 7.2.16 The fn:zero-or-one, fn:one-or-more, and fn:exactly-one functions^FS

15.2.2 fn:one-or-more

```
fn:one-or-more($arg as item()*) as item()+
```

Summary: Returns `$arg` if it contains one or more items. Otherwise, raises an error [err:FORG0004].

For detailed type semantics, see Section 7.2.16 The fn:zero-or-one, fn:one-or-more, and fn:exactly-one functions^FS

15.2.3 fn:exactly-one

```
fn:exactly-one($arg as item()*) as item()
```

Summary: Returns `$arg` if it contains exactly one item. Otherwise, raises an error [err:FORG0005].

For detailed type semantics, see Section 7.2.16 The fn:zero-or-one, fn:one-or-more, and fn:exactly-one functions^FS

15.3 Equals, Union, Intersection and Except

Function	Meaning
`fn:deep-equal`	Returns `true` if the two arguments have items that compare equal in corresponding positions.
`op:union`	Returns the union of the two sequence arguments, eliminating duplicates.
`op:intersect`	Returns the intersection of the two sequence arguments, eliminating duplicates.
`op:except`	Returns the difference of the two sequence arguments, eliminating duplicates.

As in the previous sections, for the illustrative examples below, assume an XQuery or transformation operating on a Purchase Order document containing a number of line-item elements. The variables `$item1`, `$item2`, etc. are bound to individual line-item nodes in the sequence. We use sequences of these nodes in some of the examples below.

15.3.1 fn:deep-equal

```
fn:deep-equal($parameter1 as item()*, $parameter2 as item()*) as xs:boolean
fn:deep-equal($parameter1 as item()*,
              $parameter2 as item()*,
              $collation  as string) as xs:boolean
```

Summary: This function assesses whether two sequences are deep-equal to each other. To be deep-equal, they must contain items that are pairwise deep-equal; and for two items to be deep-equal, they must either be atomic values that compare equal, or nodes of the same kind, with the same name, whose children are deep-equal. This is defined in more detail below. The `$collation` argument identifies a collation which is used at all levels of recursion when strings are compared (but not when names are compared), according to the rules in **7.3.1 Collations**.

If the two sequences are both empty, the function returns `true`.

If the two sequences are of different lengths, the function returns `false`.

If the two sequences are of the same length, the function returns `true` if and only if every item in the sequence `$parameter1` is deep-equal to the item at the same position in the sequence `$parameter2`. The rules for deciding whether two items are deep-equal follow.

Call the two items `$i1` and `$i2` respectively.

If `$i1` and `$i2` are both atomic values, they are deep-equal if and only if `($i1 eq $i2)` is `true`, or if both values are `NaN`. If the `eq` operator is not defined for `$i1` and `$i2`, the function returns `false`.

If one of the pair `$i1` or `$i2` is an atomic value and the other is a node, the function returns `false`.

If `$i1` and `$i2` are both nodes, they are compared as described below:

If the two nodes are of different kinds, the result is `false`.

If the two nodes are both document nodes then they are deep-equal if and only if the sequence `$i1/(*|text())` is deep-equal to the sequence `$i2/(*|text())`.

If the two nodes are both element nodes then they are deep-equal if and only if all of the following conditions are satisfied:

1. the two nodes have the same name, that is `(node-name($i1) eq node-name($i2))`.
2. the two nodes are both annotated as having simple content or both nodes are annotated as having complex content.
3. the two nodes have the same number of attributes, and for every attribute `$a1` in `$i1/@*` there exists an attribute `$a2` in `$i2/@*` such that `$a1` and `$a2` are deep-equal.
4. One of the following conditions holds:
    - Both element nodes have a type annotation that is simple content, and the typed value of `$i1` is deep-equal to the typed value of `$i2`.
    - Both element nodes have a type annotation that is complex content with elementOnly content, and each child element of `$i1` is deep-equal to the corresponding child element of `$i2`.
    - Both element nodes have a type annotation that is complex content with mixed content, and the sequence `$i1/(*|text())` is deep-equal to the sequence `$i2/(*|text())`.
    - Both element nodes have a type annotation that is complex content with empty content.

If the two nodes are both attribute nodes then they are deep-equal if and only if both the following conditions are satisfied:

1. the two nodes have the same name, that is `(node-name($i1) eq node-name($i2))`.
2. the typed value of `$i1` is deep-equal to the typed value of `$i2`.

If the two nodes are both processing instruction nodes, then they are deep-equal if and only if both the following conditions are satisfied:

1. the two nodes have the same name, that is `(fn:node-name($i1) eq fn:node-name($i2))`.
2. the string value of `$i1` is equal to the string value of `$i2`.

If the two nodes are both namespace nodes, then they are deep-equal if and only if both the following conditions are satisfied:

1. the two nodes either have the same name or are both nameless, that is `fn:deep-equal(fn:node-name($i1), fn:node-name($i2))`.
2. the string value of `$i1` is equal to the string value of `$i2` when compared using the Unicode codepoint collation.

If the two nodes are both text nodes or comment nodes, then they are deep-equal if and only if their string-values are equal.

Notes:

The two nodes are not required to have the same type annotation, and they are not required to have the same in-scope namespaces. They may also differ in their parent, their base URI, and the values returned by the `is-id` and `is-idrefs` accessors (see Section 5.5 is-id Accessor*DM* and Section 5.6 is-idrefs Accessor*DM*). The order of children is significant, but the order of attributes is insignificant.

The contents of comments and processing instructions are significant only if these nodes appear directly as items in the two sequences being compared. The content of a comment or processing instruction that appears as a descendant of an item in one of the sequences being compared does not affect the result. However, the presence of a comment or processing instruction, if it causes a text node to be split into two text nodes, may affect the result.

The result of `fn:deep-equal(1, current-dateTime())` is `false`; it does not raise an error.

15.3.1.1 Examples

```
let $at := <attendees> <name last='Parker'
                             first='Peter'/> <name last='Barker' first='Bob'/>
                             <name last='Parker' first='Peter'/> </attendees>
```

- `fn:deep-equal($at, $at/*)` returns `false`.
- `fn:deep-equal($at/name[1], $at/name[2])` returns `false`.
- `fn:deep-equal($at/name[1], $at/name[3])` returns `true`.
- `fn:deep-equal($at/name[1], 'Peter Parker')` returns `false`.

15.3.2 op:union

```
op:union($parameter1 as node()*, $parameter2 as node()*) as node()*
```

Summary: Constructs a sequence containing every node that occurs in the values of either `$parameter1` or `$parameter2`, eliminating duplicate nodes. Nodes are returned in document order. Two nodes are duplicates if they are `op:is-same-node()`.

If either operand is the empty sequence, a sequence is returned containing the nodes in the other operand in document order after eliminating duplicates.

For detailed type semantics, see Section 7.2.14 The op:union, op:intersect, and op:except operators*FS*

This function backs up the "union" or "|" operator.

15.3.2.1 Examples

Assume `$seq1 = ($item1, $item2)`, `$seq2 = ($item1, $item2)` and `$seq3 = ($item2, $item3)`.

- `op:union($seq1, $seq1)` returns the sequence `($item1, $item2)`.
- `op:union($seq2, $seq3)` returns the sequence consisting of `($item1, $item2, $item3)`.

15.3.3 op:intersect

```
op:intersect($parameter1 as node()*, $parameter2 as node()*) as node()*
```

Summary: Constructs a sequence containing every node that occurs in the values of both `$parameter1` and `$parameter2`, eliminating duplicate nodes. Nodes are returned in document order.

If either operand is the empty sequence, the empty sequence is returned.

Two nodes are duplicates if they are `op:is-same-node()`.

For detailed type semantics, see Section 7.2.14 The op:union, op:intersect, and op:except operators[FS].

This function backs up the "intersect" operator.

15.3.3.1 Examples

Assume `$seq1 = ($item1, $item2)`, `$seq2 = ($item1, $item2)` and `$seq3 = ($item2, $item3)`.

- `op:intersect($seq1, $seq1)` returns the sequence `($item1, $item2)`.
- `op:intersect($seq2, $seq3)` returns the sequence `($item2)`.

15.3.4 op:except

```
op:except($parameter1 as node()*, $parameter2 as node()*) as node()*
```

Summary: Constructs a sequence containing every node that occurs in the value of `$parameter1`, but not in the value of `$parameter2`, eliminating duplicate nodes. Nodes are returned in document order.

If `$parameter1` is the empty sequence, the empty sequence is returned. If `$parameter2` is the empty sequence, a sequence is returned containing the nodes in `$parameter1` in document order after eliminating duplicates.

Two nodes are duplicates if they are `op:is-same-node()`.

For detailed type semantics, see Section 7.2.14 The op:union, op:intersect, and op:except operators[FS].

This function backs up the "except" operator.

15.3.4.1 Examples

Assume `$seq1 = ($item1, $item2)`, `$seq2 = ($item1, $item2)` and `$seq3 = ($item2, $item3)`.

- `op:except($seq1, $seq2)` returns the empty sequence.
- `op:except($seq2, $seq3)` returns the sequence `($item1)`.

15.4 Aggregate Functions

Aggregate functions take a sequence as argument and return a single value computed from values in the sequence. Except for `fn:count`, the sequence must consist of values of a single type or one if its subtypes, or they must be numeric. `xs:untypedAtomic` values are permitted in the input sequence and handled by special conversion rules. The type of the items in the sequence must also support certain operations.

Function	Meaning
`fn:count`	Returns the number of items in a sequence.
`fn:avg`	Returns the average of a sequence of values.
`fn:max`	Returns the maximum value from a sequence of comparable values.
`fn:min`	Returns the minimum value from a sequence of comparable values.
`fn:sum`	Returns the sum of a sequence of values.

15.4.1 fn:count

```
fn:count($arg as item()*) as xs:integer
```

Summary: Returns the number of items in the value of `$arg`.

Returns 0 if `$arg` is the empty sequence.

15.4.1.1 Examples

Assume `$seq1 = ($item1, $item2)` and `$seq3 = ()`, the empty sequence.

- `fn:count($seq1)` returns `2`.
- `fn:count($seq3)` returns `0`.

Assume `$seq2 = (98.5, 98.3, 98.9)`.

- `fn:count($seq2)` returns `3`.
- `fn:count($seq2[. > 100])` returns `0`.

15.4.2 fn:avg

```
fn:avg($arg as xs:anyAtomicType*) as xs:anyAtomicType?
```

Summary: Returns the average of the values in the input sequence `$arg`, that is, the sum of the values divided by the number of values.

If `$arg` is the empty sequence, the empty sequence is returned.

If `$arg` contains values of type `xs:untypedAtomic` they are cast to `xs:double`.

Duration values must either all be `xs:yearMonthDuration` values or must all be `xs:dayTimeDuration` values. For numeric values, the numeric promotion rules defined in **6.2 Operators on Numeric Values** are used to promote all values to a single common type. After these operations, `$arg` must contain items of a single type, which must be one of the four numeric types, `xs:yearMonthDuration` or `xs:dayTimeDuration` or one if its subtypes.

If the above conditions are not met, then a type error is raised [err:FORG0006].

Otherwise, returns the average of the values as `sum($arg) div count($arg)`; but the implementation may use an otherwise equivalent algorithm that avoids arithmetic overflow.

For detailed type semantics, see Section 7.2.10 The fn:min, fn:max, fn:avg, and fn:sum functions^FS.

15.4.2.1 Examples

Assume `$d1 = xs:yearMonthDuration("P20Y")` and `$d2 = xs:yearMonthDuration("P10M")` and `$seq3 = (3, 4, 5)`.

- `fn:avg($seq3)` returns `4.0`.
- `fn:avg(($d1, $d2))` returns a `yearMonthDuration` with value `125` months.
- `fn:avg(($d1, $seq3))` raises a type error [err:FORG0006].
- `fn:avg(())` returns `()`.
- `fn:avg((xs:float('INF'), xs:float('-INF')))` returns `NaN`.
- `fn:avg(($seq3, xs:float('NaN')))` returns `NaN`.

15.4.3 fn:max

```
fn:max($arg as xs:anyAtomicType*) as xs:anyAtomicType?
fn:max($arg as xs:anyAtomicType*, $collation as string) as xs:anyAtomicType?
```

Summary: Selects an item from the input sequence `$arg` whose value is greater than or equal to the value of every other item in the input sequence. If there are two or more such items, then the specific item whose value is returned is ·implementation dependent·.

The following rules are applied to the input sequence:

- Values of type `xs:untypedAtomic` in `$arg` are cast to `xs:double`.
- Numeric values are converted to their least common type reachable by a combination of type promotion and subtype substitution. See Section B.1 Type Promotion^XP and Section B.2 Operator Mapping^XP.
- Values of type `xs:anyURI` are cast to `xs:string`

The items in the resulting sequence may be reordered in an arbitrary order. The resulting sequence is referred to below as the converted sequence. This function returns an item from the converted sequence rather than the input sequence.

If the converted sequence is empty, the empty sequence is returned.

All items in the converted sequence must be derived from a single base type for which the `le` operator is defined. In addition, the values in the sequence must have a total order. If date/time values do not have a timezone, they are considered to have the implicit timezone provided by the dynamic context for the purpose of comparison. Duration values must either all be `xs:yearMonthDuration` values or must all be `xs:dayTimeDuration` values.

If any of these conditions is not met, then a type error is raised [err:FORG0006].

If the converted sequence contains the value `NaN`, the value `NaN` is returned.

If the items in the converted sequence are of type `xs:string` or types derived by restriction from `xs:string`, then the determination of the item with the smallest value is made according to the collation that is used. If the type of the items in the converted sequence is not `xs:string` and `$collation` is specified, the collation is ignored.

The collation used by the invocation of this function is determined according to the rules in **7.3.1 Collations**.

Otherwise, the result of the function is the result of the expression:

```
    if (every $v in $c satisfies $c[1] ge $v)
    then $c[1]
    else fn:max(fn:subsequence($c, 2))
```

evaluated with `$collation` as the default collation if specified, and with `$c` as the converted sequence.

For detailed type semantics, see Section 7.2.10 The fn:min, fn:max, fn:avg, and fn:sum functions^FS.

Notes:

If the converted sequence contains exactly one value then that value is returned.

The default type when the `fn:max` function is applied to `xs:untypedAtomic` values is `xs:double`. This differs from the default type for operators such as `gt`, and for sorting in XQuery and XSLT, which is `xs:string`.

15.4.3.1 Examples

- `fn:max((3,4,5))` returns `5`.
- `fn:max((5, 5.0e0))` returns `5.0e0`.
- `fn:max((3,4,"Zero"))` raises a type error [err:FORG0006].
- `fn:max((fn:current-date(), xs:date("2001-01-01")))` typically returns the current date.
- `fn:max(("a", "b", "c"))` returns "c" under a typical default collation.

15.4.4 fn:min

```
fn:min($arg as xs:anyAtomicType*) as xs:anyAtomicType?
fn:min($arg as xs:anyAtomicType*, $collation as string) as xs:anyAtomicType?
```

Summary: selects an item from the input sequence `$arg` whose value is less than or equal to the value of every other item in the input sequence. If there are two or more such items, then the specific item whose value is returned is ·implementation dependent·.

The following rules are applied to the input sequence:

- Values of type `xs:untypedAtomic` in `$arg` are cast to `xs:double`.
- Numeric values are converted to their least common type reachable by a combination of type promotion and subtype substitution. See Section B.1 Type Promotion^XP and Section B.2 Operator Mapping^XP.
- Values of type `xs:anyURI` are cast to `xs:string`

The items in the resulting sequence may be reordered in an arbitrary order. The resulting sequence is referred to below as the converted sequence. This function returns an item from the converted sequence rather than the input sequence.

If the converted sequence is empty, the empty sequence is returned.

All items in the converted sequence must be derived from a single base type for which the `le` operator is defined. In addition, the values in the sequence must have a total order. If date/time values do not have a timezone, they are considered to have the implicit timezone provided by the dynamic context for the purpose of comparison. Duration values must either all be `xs:yearMonthDuration` values or must all be `xs:dayTimeDuration` values.

If any of these conditions is not met, a type error is raised [err:FORG0006].

If the converted sequence contains the value `NaN`, the value `NaN` is returned.

If the items in the converted sequence are of type `xs:string` or types derived by restriction from `xs:string`, then the determination of the item with the smallest value is made according to the collation that is used. If the type of the items in the converted sequence is not `xs:string` and `$collation` is specified, the collation is ignored.

The collation used by the invocation of this function is determined according to the rules in **7.3.1 Collations**.

Otherwise, the result of the function is the result of the expression:

```
    if (every $v in $c satisfies $c[1] le $v)
    then $c[1]
    else fn:min(fn:subsequence($c, 2))
```

evaluated with `$collation` as the default collation if specified, and with `$c` as the converted sequence.

For detailed type semantics, see Section 7.2.10 The fn:min, fn:max, fn:avg, and fn:sum functions*FS*.

Notes:

If the converted sequence contains exactly one value then that value is returned.

The default type when the `fn:min` function is applied to `xs:untypedAtomic` values is `xs:double`. This differs from the default type for operators such as `lt`, and for sorting in XQuery and XSLT, which is `xs:string`.

15.4.4.1 Examples

- `fn:min((3,4,5))` returns `3`.
- `fn:min((5, 5.0e0))` returns `5.0e0`.
- `fn:min((3,4,"Zero"))` raises a type error [err:FORG0006].
- `fn:min((xs:float(0.0E0), xs:float(-0.0E0)))` can return either positive or negative zero. [XML Schema Part 2: Datatypes Second Edition] does not distinguish between the values positive zero and negative zero. The result is ·implementation dependent·.
- `fn:min((fn:current-date(), xs:date("2001-01-01")))` typically returns `xs:date("2001-01-01")`.
- `fn:min(("a", "b", "c"))` returns "a" under a typical default collation.

15.4.5 fn:sum

```
fn:sum($arg as xs:anyAtomicType*) as xs:anyAtomicType
```

```
fn:sum($arg  as xs:anyAtomicType*,
       $zero as xs:anyAtomicType?) as xs:anyAtomicType?
```

Summary: Returns a value obtained by adding together the values in `$arg`. If `$zero` is not specified, then the value returned for an empty sequence is the `xs:integer` value 0. If `$zero` is specified, then the value returned for an empty sequence is `$zero`.

Any values of type `xs:untypedAtomic` in `$arg` are cast to `xs:double`. The items in the resulting sequence may be reordered in an arbitrary order. The resulting sequence is referred to below as the converted sequence.

If the converted sequence is empty, then the single-argument form of the function returns the `xs:integer` value `0`; the two-argument form returns the value of the argument `$zero`.

If the converted sequence contains the value `NaN`, `NaN` is returned.

All items in `$arg` must be numeric or derived from a single base type. In addition, the type must support addition. Duration values must either all be `xs:yearMonthDuration` values or must all be `xs:dayTimeDuration` values. For numeric values, the numeric promotion rules defined in **6.2 Operators on Numeric Values** are used to promote all values to a single common type. The sum of a sequence of integers will therefore be an integer, while the sum of a numeric sequence that includes at least one xs:double will be an xs:double.

If the above conditions are not met, a type error is raised [err:FORG0006].

Otherwise, the result of the function, using the second signature, is the result of the expression:

```
if (fn:count($c) eq 0) then
    $zero
else if (fn:count($c) eq 1) then
    $c[1]
else
    $c[1] + fn:sum(subsequence($c, 2))
```

where `$c` is the converted sequence.

The result of the function, using the first signature, is the result of the expression: `fn:sum($arg, 0)`.

For detailed type semantics, see Section 7.2.10 The fn:min, fn:max, fn:avg, and fn:sum functions^FS.

Notes:

The second argument allows an appropriate value to be defined to represent the sum of an empty sequence. For example, when summing a sequence of durations it would be appropriate to return a zero-length duration of the appropriate type. This argument is necessary because a system that does dynamic typing cannot distinguish "an empty sequence of integers", for example, from "an empty sequence of durations".

If the converted sequence contains exactly one value then that value is returned.

15.4.5.1 Examples

Assume:

```
$d1 = xs:yearMonthDuration("P20Y")
$d2 = xs:yearMonthDuration("P10M")
$seq1 = ($d1, $d2)
$seq3 = (3, 4, 5)
```

- `fn:sum(($d1, $d2))` returns an `xs:yearMonthDuration` with a value of `250` months.
- `fn:sum($seq1[. < xs:yearMonthDuration('P3M')], xs:yearMonthDuration('P0M'))` returns an `xs:yearMonthDuration` with a value of `0` months.
- `fn:sum($seq3)` returns `12`.
- `fn:sum(())` returns `0`.
- `fn:sum((),())` returns `()`.
- `fn:sum((1 to 100)[.<0], 0)` returns `0`.
- `fn:sum(($d1, 9E1))` raises an error [err:FORG0006].

15.5 Functions and Operators that Generate Sequences

Function	Meaning
`op:to`	Returns the sequence containing every `xs:integer` between the values of the operands.
`fn:id`	Returns the sequence of element nodes having an ID value matching the one or more of the supplied IDREF values.
`fn:idref`	Returns the sequence of element or attribute nodes with an IDREF value matching one or more of the supplied ID values.
`fn:doc`	Returns a document node retrieved using the specified URI.
`fn:doc-available`	Returns `true` if a document node can be retrieved using the specified URI.
`fn:collection`	Returns a sequence of nodes retrieved using the specified URI or the nodes in the default collection.

15.5.1 op:to

```
op:to($firstval as xs:integer, $lastval as xs:integer) as xs:integer*
```

Summary: Returns the sequence containing every `xs:integer` whose value is between the value of `$firstval` (inclusive) and the value of `$lastval` (inclusive), in monotonic order. If the value of the first operand is greater than the value of the second, the empty sequence is returned. If the values of the two operands are equal, a sequence containing a single `xs:integer` equal to the value is returned.

This function backs up the "to" operator.

15.5.1.1 Examples

- `1 to 3` returns `(1, 2, 3)`
- `3 to 1` returns `()`
- `5 to 5` returns `5`

15.5.2 fn:id

```
fn:id($arg as xs:string*) as element()*
```

```
fn:id($arg as xs:string*, $node as node()) as element()*
```

Summary: Returns the sequence of element nodes that have an `ID` value matching the value of one or more of the `IDREF` values supplied in `$arg` .

Note:

This function does not have the desired effect when searching a document in which elements of type `xs:ID` are used as identifiers. To preserve backwards compatibility, a new function `fn:element-with-id` is therefore being introduced; it behaves the same way as `fn:id` in the case of ID-valued attributes.

The function returns a sequence, in document order with duplicates eliminated, containing every element node `E` that satisfies all the following conditions:

1. `E` is in the target document. The target document is the document containing `$node`, or the document containing the context item (`.`) if the second argument is omitted. The behavior of the function if `$node` is omitted is exactly the same as if the context item had been passed as `$node`. If `$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:FODC0001] is raised. If the second argument is the context item, or is omitted, the following errors may be raised: if there is no context item, [err:XPDY0002]XP; if the context item is not a node [err:XPTY0004]XP.
2. `E` has an `ID` value equal to one of the candidate `IDREF` values, where:
    - An element has an `ID` value equal to `V` if either or both of the following conditions are true:
        - The `is-id` property (See Section 5.5 is-id AccessorDM.) of the element node is true, and the typed value of the element node is equal to V under the rules of the `eq` operator using the Unicode code point collation (`http://www.w3.org/2005/xpath-functions/collation/codepoint`).
        - The element has an attribute node whose `is-id` property (See Section 5.5 is-id AccessorDM.) is true and whose typed value is equal to `V` under the rules of the `eq` operator using the Unicode code point collation (`http://www.w3.org/2005/xpath-functions/collation/codepoint`).
    - Each `xs:string` in `$arg` is treated as a whitespace-separated sequence of tokens, each token acting as an `IDREF`. These tokens are then included in the list of candidate `IDREF` values. If any of the tokens is not a lexically valid `IDREF` (that is, if it is not lexically an `xs:NCName`), it is ignored. Formally, the candidate `IDREF` values are the strings in the sequence given by the expression:

    ```
    for $s in $arg return fn:tokenize(fn:normalize-space($s), ' ')
                      [. castable as xs:IDREF]
    ```
3. If several elements have the same `ID` value, then `E` is the one that is first in document order.

Notes:

If the data model is constructed from an Infoset, an attribute will have the `is-id` property if the corresponding attribute in the Infoset had an attribute type of `ID`: typically this means the attribute was declared as an `ID` in a DTD.

If the data model is constructed from a PSVI, an element or attribute will have the `is-id` property if its typed value is a single atomic value of type `xs:ID` or a type derived by restriction from `xs:ID`.

No error is raised in respect of a candidate `IDREF` value that does not match the `ID` of any element in the document. If no candidate `IDREF` value matches the `ID` value of any element, the function returns the empty sequence.

It is not necessary that the supplied argument should have type `xs:IDREF` or `xs:IDREFS`, or that it should be derived from a node with the `is-idrefs` property.

An element may have more than one `ID` value. This can occur with synthetic data models or with data models constructed from a PSVI where the element and one of its attributes are both typed as `xs:ID`.

If the source document is well-formed but not valid, it is possible for two or more elements to have the same `ID` value. In this situation, the function will select the first such element.

It is also possible in a well-formed but invalid document to have an element or attribute that has the is-id property but whose value does not conform to the lexical rules for the `xs:ID` type. Such a node will never be selected by this function.

15.5.3 fn:idref

```
fn:idref($arg as xs:string*) as node()*
```

```
fn:idref($arg as xs:string*, $node as node()) as node()*
```

Summary: Returns the sequence of element or attribute nodes with an `IDREF` value matching the value of one or more of the `ID` values supplied in `$arg`.

The function returns a sequence, in document order with duplicates eliminated, containing every element or attribute node `$N` that satisfies all the following conditions:

1. `$N` is in the target document. The target document is the document containing `$node` or the document containing the context item (`.`) if the second argument is omitted. The behavior of the function if `$node` is omitted is exactly the same as if the context item had been passed as `$node`. If `$node`, or the context item if the second argument is omitted, is a node in a tree whose root is not a document node [err:FODC0001] is raised. If the second argument is the context item, or is omitted, the following errors may be raised: if there is no context item [err:XPDY0002]XP; if the context item is not a node [err:XPTY0004]XP.
2. `$N` has an `IDREF` value equal to one of the candidate `ID` values, where:
   - A node `$N` has an `IDREF` value equal to `V` if both of the following conditions are true:
     - The `is-idrefs` property (See Section 5.6 is-idrefs AccessorDM.)of `$N` is `true`
     - The sequence `fn:tokenize(fn:normalize-space(fn:string($N)), ' ')` contains a string that is equal to `V` under the rules of the `eq` operator using the Unicode code point collation (`http://www.w3.org/2005/xpath-functions/collation/codepoint`).
   - Each `xs:string` in `$arg` is parsed as if it were of lexically of type `xs:ID`. These `xs:string`s are then included in the list of candidate `xs:ID`s. If any of the strings in `$arg` is not a lexically valid `xs:ID` (that is, if it is not lexically an `xs:NCName`), it is ignored. More formally, the candidate `ID` values are the strings in the sequence

     ```
     $arg[. castable as xs:NCName]
     ```

Notes:

An element or attribute typically acquires the `is-idrefs` property by being validated against the schema type `xs:IDREF` or `xs:IDREFS`, or (for attributes only) by being described as of type `IDREF` or `IDREFS` in a DTD.

No error is raised in respect of a candidate `ID` value that does not match the `IDREF` value of any element or attribute in the document. If no candidate `ID` value matches the `IDREF` value of any element or attribute, the function returns the empty sequence.

It is possible for two or more nodes to have an `IDREF` value that matches a given candidate `ID` value. In this situation, the function will return all such nodes. However, each matching node will be returned at most once, regardless how many candidate `ID` values it matches.

It is possible in a well-formed but invalid document to have a node whose `is-idrefs` property is true but that does not conform to the lexical rules for the `xs:IDREF` type. The effect of the above rules is that ill-formed candidate `ID` values and ill-formed `IDREF` values are ignored.

If the data model is constructed from a PSVI, the typed value of a node that has the `is-idrefs` property will contain at least one atomic value of type `xs:IDREF` (or a type derived by restriction from `xs:IDREF`). It may also contain atomic values of other types. These atomic values are treated as candidate `ID` values if their lexical form is valid as an `xs:NCName`, and they are ignored otherwise.

15.5.4 fn:doc

```
fn:doc($uri as xs:string?) as document-node()?
```

Summary: Retrieves a document using a URI supplied as an `xs:string`, and returns the corresponding document node.

If `$uri` is the empty sequence, the result is an empty sequence.

If `$uri` is not a valid URI, an error **may** be raised [err:FODC0005].

If `$uri` is a relative URI reference, it is resolved relative to the value of the base URI property from the static context. The resulting absolute URI is promoted to an `xs:string`.

If the **Available documents** described in Section 2.1.2 Dynamic Context^XP provides a mapping from this string to a document node, the function returns that document node.

If the **Available documents** provides no mapping for the string, an error is raised [err:FODC0005].

The URI may include a fragment identifier.

By default, this function is ·stable·. Two calls on this function return the same document node if the same URI Reference (after resolution to an absolute URI Reference) is supplied to both calls. Thus, the following expression (if it does not raise an error) will always be true:

```
doc("foo.xml") is doc("foo.xml")
```

However, for performance reasons, implementations may provide a user option to evaluate the function without a guarantee of stability. The manner in which any such option is provided is implementation-defined. If the user has not selected such an option, a call of the function must either return a stable result or must raise an error: [err:FODC0003].

For detailed type semantics, see Section 7.2.5 The fn:collection and fn:doc functions^FS.

Note:

If `$uri` is read from a source document, it is generally appropriate to resolve it relative to the base URI property of the relevant node in the source document. This can be achieved by calling the `fn:resolve-uri` function, and passing the resulting absolute URI as an argument to the `fn:doc` function.

If two calls to this function supply different absolute URI References as arguments, the same document node may be returned if the implementation can determine that the two arguments refer to the same resource.

By defining the semantics of this function in terms of a string-to-document-node mapping in the dynamic context, the specification is acknowledging that the results of this function are outside the purview of the language specification itself, and depend entirely on the run-time environment in which the expression is evaluated. This run-time environment includes not only an unpredictable collection of resources ("the web"), but configurable machinery for locating resources and turning their contents into document nodes within the XPath data model. Both the set of resources that are reachable, and the mechanisms by which those resources are parsed and validated, are ·implementation dependent·.

One possible processing model for this function is as follows. The resource identified by the URI Reference is retrieved. If the resource cannot be retrieved, an error is raised [err:FODC0002]. The data resulting from the retrieval action is then parsed as an XML document and a tree is constructed in accordance with the [XQuery 1.0 and XPath 2.0 Data Model]. If the top-level media type is known and is "text", the content is parsed in the same way as if the media type were text/xml; otherwise, it is parsed in the same way as if the media type were application/xml. If the contents cannot be parsed successfully, an error is raised [err:FODC0002]. Otherwise, the result of the function is the document node at the root of the resulting tree. This tree is then optionally validated against a schema.

Various aspects of this processing are ·implementation-defined·. Implementations may provide external configuration options that allow any aspect of the processing to be controlled by the user. In particular:

- The set of URI schemes that the implementation recognizes is implementation-defined. Implementations may allow the mapping of URIs to resources to be configured by the user, using mechanisms such as catalogs or user-written URI handlers.
- The handling of non-XML media types is implementation-defined. Implementations may allow instances of the data model to be constructed from non-XML resources, under user control.
- It is ·implementation-defined· whether DTD validation and/or schema validation is applied to the source document.
- Implementations may provide user-defined error handling options that allow processing to continue following an error in retrieving a resource, or in parsing and validating its content. When errors have been handled in this way, the function may return either an empty sequence, or a fallback document provided by the error handler.
- Implementations may provide user options that relax the requirement for the function to return stable results.

15.5.5 fn:doc-available

```
fn:doc-available($uri as xs:string?) as xs:boolean
```

Summary: The function returns true if and only if the function call `fn:doc($uri)` would return a document node.

If `$uri` is an empty sequence, this function returns `false`.

If a call on `fn:doc($uri)` would return a document node, this function returns `true`.

If `$uri` is not a valid URI according to the rules applied by the implementation of `fn:doc`, an error is raised [err:FODC0005].

Otherwise, this function returns `false`.

If this function returns `true`, then calling `fn:doc($uri)` within the same ·execution scope· must return a document node. However, if non-stable processing has been selected for the `fn:doc` function, this guarantee is lost.

15.5.6 fn:collection

```
fn:collection() as node()*
```

```
fn:collection($arg as xs:string?) as node()*
```

Summary: This function takes an `xs:string` as argument and returns a sequence of nodes obtained by interpreting `$arg` as an `xs:anyURI` and resolving it according to the mapping specified in **Available collections** described in Section C.2 Dynamic Context Components^XP. If **Available collections** provides a mapping from this string to a sequence of nodes, the function returns that sequence. If **Available collections** maps the string to an empty sequence, then the function returns an empty sequence. If **Available collections** provides no mapping for the string, an error is raised [err:FODC0004]. If `$arg` is not specified, the function returns the sequence of the nodes in the default collection in the dynamic context. See Section C.2 Dynamic Context Components^XP. If the value of the default collection is undefined an error is raised [err:FODC0002].

If the `$arg` is a relative `xs:anyURI`, it is resolved against the value of the base-URI property from the static context. If `$arg` is not a valid `xs:anyURI`, an error is raised [err:FODC0004].

If `$arg` is the empty sequence, the function behaves as if it had been called without an argument. See above.

By default, this function is ·stable·. This means that repeated calls on the function with the same argument will return the same result. However, for performance reasons, implementations may provide a user option to evaluate the function without a guarantee of stability. The manner in which any such option is provided is ·implementation-defined·. If the user has not selected such an option, a call to this function must either return a stable result or must raise an error: [err:FODC0003].

For detailed type semantics, see Section 7.2.5 The fn:collection and fn:doc functions^FS.

> **Note:**
>
> This function provides a facility for users to work with a collection of documents which may be contained in a directory or rows of a Relational table or other implementation-specific construct. An implementation may also use external variables to identify external resources, but `fn:collection()` provides functionality not provided by external variables. Specifying resources using URIs is useful because URIs are dynamic, can be parameterized, and do not rely on an external environment.

15.5.7 fn:element-with-id

```
fn:element-with-id($arg as xs:string*) as element()*
```

```
fn:element-with-id($arg as xs:string*, $node as node()) as element()*
```

Summary: Returns the sequence of element nodes that have an `ID` value matching the value of one or more of the `IDREF` values supplied in `$arg`.

> **Note:**
>
> The `fn:id` function does not have the desired effect when searching a document in which elements of type `xs:ID` are used as identifiers. To preserve backwards compatibility, this function `fn:element-with-id` is therefore being introduced; it behaves the same way as `fn:id` in the case of ID-valued attributes.
>
> Unless otherwise specified in the conformance rules for a host language, implementation of this function is optional. Introduction of the function by means of an erratum therefore does not make existing implementations non-conformant.

The function returns a sequence, in document order with duplicates eliminated, containing every element node `E` that satisfies all the following conditions:

1. `E` is in the target document. The target document is the document containing `$node`, or the document containing the context item (`.`) if the second argument is omitted. The behavior of the function if `$node` is omitted is exactly the same as if the context item had been passed as `$node`. If `$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:FODC0001] is raised. If the second argument is the context item, or is omitted, the following errors may be raised: if there is no context item, [err:XPDY0002]XP; if the context item is not a node [err:XPTY0004]XP.
2. `E` has an `ID` value equal to one of the candidate `IDREF` values, where:
   - An element has an `ID` value equal to `V` if either or both of the following conditions are true:
     - The element has a child element node whose `is-id` property (See Section 5.5 is-id AccessorDM.) is true, and whose typed value is equal to `V` under the rules of the `eq` operator using the Unicode code point collation.
     - The element has an attribute node whose `is-id` property (See Section 5.5 is-id AccessorDM.) is true, and whose typed value is equal to `V` under the rules of the `eq` operator using the Unicode code point collation.
   - Each `xs:string` in `$arg` is treated as a whitespace-separated sequence of tokens, each acting as an `IDREF`. These tokens are then included in the list of candidate `IDREF` values. If any of the tokens is not a lexically valid `IDREF` (that is, if it is not lexically an `xs:NCName`), it is ignored. Formally, the candidate `IDREF` values are the strings in the sequence given by the expression:

     ```
     for $s in $arg return fn:tokenize(fn:normalize-space($s), ' ')[. castable as xs:IDREF]
     ```
3. If several elements have the same `ID` value, then `E` is the one that is first in document order.

> **Notes:**
>
> See the Notes for the `fn:id` function, all of which apply equally to this function.

16 Context Functions

The following functions are defined to obtain information from the dynamic context.

Function	Meaning

fn:position	Returns the position of the context item within the sequence of items currently being processed.
fn:last	Returns the number of items in the sequence of items currently being processed.
fn:current-dateTime	Returns the current `xs:dateTime`.
fn:current-date	Returns the current `xs:date`.
fn:current-time	Returns the current `xs:time`.
fn:implicit-timezone	Returns the value of the implicit timezone property from the dynamic context.
fn:default-collation	Returns the value of the default collation property from the static context.
fn:static-base-uri	Returns the value of the Base URI property from the static context.

16.1 fn:position

```
fn:position() as xs:integer
```

Summary: Returns the context position from the dynamic context. (See Section C.2 Dynamic Context Components^XP.) If the context item is undefined, an error is raised: [err:XPDY0002]^XP.

16.2 fn:last

```
fn:last() as xs:integer
```

Summary: Returns the context size from the dynamic context. (See Section C.2 Dynamic Context Components^XP.) If the context item is undefined, an error is raised: [err:XPDY0002]^XP.

16.3 fn:current-dateTime

```
fn:current-dateTime() as xs:dateTime
```

Summary: Returns the current dateTime (with timezone) from the dynamic context. (See Section C.2 Dynamic Context Components^XP.) This is an `xs:dateTime` that is current at some time during the evaluation of a query or transformation in which `fn:current-dateTime()` is executed. This function is ·stable·. The precise instant during the query or transformation represented by the value of `fn:current-dateTime()` is ·implementation dependent·.

16.3.1 Examples

- `fn:current-dateTime()` returns an `xs:dateTime` corresponding to the current date and time. For example, an invocation of `fn:current-dateTime()` might return `2004-05-12T18:17:15.125Z` corresponding to the current time on May 12, 2004 in timezone `Z`.

16.4 fn:current-date

```
fn:current-date() as xs:date
```

Summary: Returns `xs:date(fn:current-dateTime())`. This is an `xs:date` (with timezone) that is current at some time during the evaluation of a query or transformation in which `fn:current-date()` is executed. This function is ·stable·. The precise instant during the query or transformation represented by the value of `fn:current-date()` is ·implementation dependent·.

16.4.1 Examples

- `fn:current-date()` returns an `xs:date` corresponding to the current date and time. For example, an invocation of `fn:current-date()` might return `2004-05-12+01:00`.

16.5 fn:current-time

```
fn:current-time() as xs:time
```

Summary: Returns `xs:time(fn:current-dateTime())`. This is an `xs:time` (with timezone) that is current at some time during the evaluation of a query or transformation in which `fn:current-time()` is executed. This function is ·stable·. The precise instant during the query or transformation represented by the value of `fn:current-time()` is ·implementation dependent·.

16.5.1 Examples

- `fn:current-time()` returns an `xs:time` corresponding to the current date and time. For example, an invocation of `fn:current-time()` might return `23:17:00.000-05:00`.

16.6 fn:implicit-timezone

```
fn:implicit-timezone() as xs:dayTimeDuration
```

Summary: Returns the value of the implicit timezone property from the dynamic context. Components of the dynamic context are discussed in Section C.2 Dynamic Context Components^XP.

16.7 fn:default-collation

```
fn:default-collation() as xs:string
```

Summary: Returns the value of the default collation property from the static context. Components of the static context are discussed in Section C.1 Static Context Components^XP.

> **Note:**
>
> The default collation property can never be undefined. If it is not explicitly defined, a system defined default can be invoked. If this is not provided, the Unicode code point collation (`http://www.w3.org/2005/xpath-functions/collation/codepoint`) is used.

16.8 fn:static-base-uri

```
fn:static-base-uri() as xs:anyURI?
```

Summary: Returns the value of the Base URI property from the static context. If the Base URI property is undefined, the empty sequence is returned. Components of the static context are discussed in Section C.1 Static Context Components^XP .

17 Casting

Constructor functions and cast expressions accept an expression and return a value of a given type. They both convert a source value, *SV*, of a source type, *ST*, to a target value, *TV*, of the given target type, *TT*, with identical semantics and different syntax. The name of the constructor function is the same as the name of the built-in [XML Schema Part 2: Datatypes Second Edition] datatype or the datatype defined in Section 2.6 Types^DM of [XQuery 1.0 and XPath 2.0 Data Model] (see **5.1 Constructor Functions for XML Schema Built-in Types**) or the user-derived datatype (see **5.4 Constructor Functions for User-Defined Types**) that is the target for the conversion, and the semantics are exactly the same as for a cast expression; for example," `xs:date("2003-01-01")` " means exactly the same as " `"2003-01-01"` cast as `xs:date?` ".

The cast expression takes a type name to indicate the target type of the conversion. See Section 3.10.2 Cast^XP. If the type name allows the empty sequence and the expression to be cast is the empty sequence, the empty sequence is returned. If the type name does not allow the empty sequence and the expression to be cast is the empty sequence, a type error is raised [err:XPTY0004]^XP.

Where the argument to a cast is a literal, the result of the function may be evaluated statically; if an error is encountered during such evaluation, it may be reported as a static error.

Casting from primitive type to primitive type is discussed in **17.1 Casting from primitive types to primitive types**. Casting to derived types is discussed in **17.2 Casting to derived types**. Casting from derived types is discussed in **17.3 Casting from derived types to parent types**, **17.4 Casting within a branch of the type hierarchy** and **17.5 Casting across the type hierarchy**.

When casting from `xs:string` the semantics in **17.1.1 Casting from xs:string and xs:untypedAtomic** apply, regardless of target type.

17.1 Casting from primitive types to primitive types

This section defines casting between the 19 primitive types defined in [XML Schema Part 2: Datatypes Second Edition] as well as `xs:untypedAtomic`, `xs:integer` and the two derived types of `xs:duration` (`xs:yearMonthDuration` and `xs:dayTimeDuration`). These four types are not primitive types but they are treated as primitive types in this section. The type conversions that are supported are indicated in the table below. In this table, there is a row for each primitive type with that type as the source of the conversion and there is a column for each primitive type as the target of the conversion. The intersections of rows and columns contain one of three characters: "Y" indicates that a conversion from values of the type to which the row applies to the type to which the column applies is supported; "N" indicates that there are no supported conversions from values of the type to which the row applies to the type to which the column applies; and "M" indicates that a conversion from values of the type to which the row applies to the type to which the column applies may succeed for some values in the value space and fails for others.

[XML Schema Part 2: Datatypes Second Edition] defines `xs:NOTATION` as an abstract type. Thus, casting to `xs:NOTATION` from any other type including `xs:NOTATION` is not permitted and raises [err:XPST0080]^XP. However, casting from one subtype of `xs:NOTATION` to another subtype of `xs:NOTATION` is permitted.

Casting is not supported to or from `xs:anySimpleType`. Thus, there is no row or column for this type in the table below. For any node that has not been validated or has been validated as `xs:anySimpleType`, the typed value of the node is an atomic value of type `xs:untypedAtomic`. There are no atomic values with the type annotation `xs:anySimpleType` at runtime. Casting to a type that is not atomic raises [err:XPST0051]XP.

Similarly, casting is not supported to or from `xs:anyAtomicType` and will raise error [err:XPST0080]XP. There are no atomic values with the type annotation `xs:anyAtomicType` at runtime, although this can be a statically inferred type.

If casting is attempted from an *ST* to a *TT* for which casting is not supported, as defined in the table below, a type error is raised [err:XPTY0004]XP.

In the following table, the columns and rows are identified by short codes that identify simple types as follows:

uA = xs:untypedAtomic
aURI = xs:anyURI
b64 = xs:base64Binary
bool = xs:boolean
dat = xs:date
gDay = xs:gDay
dbl = xs:double
dec = xs:decimal
dT = xs:dateTime
dTD = xs:dayTimeDuration
dur = xs:duration
flt = xs:float
hxB = xs:hexBinary
gMD = xs:gMonthDay
gMon = xs:gMonth
int = xs:integer
NOT = xs:NOTATION
QN = xs:QName
str = xs:string
tim = xs:time
gYM = xs:gYearMonth
yMD = xs:yearMonthDuration
gYr = xs:gYear

In the following table, the notation "S\T" indicates that the source ("S") of the conversion is indicated in the column below the notation and that the target ("T") is indicated in the row to the right of the notation.

S\T	uA	str	flt	dbl	dec	int	dur	yMD	dTD	dT	tim	dat	gYM	gYr	gMD	gDay	gMon	bool	b64	hxB	aURI	QN	NOT
uA	Y	Y	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	N	N
str	Y	Y	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
flt	Y	Y	Y	Y	M	M	N	N	N	N	N	N	N	N	N	N	N	Y	N	N	N	N	N
dbl	Y	Y	Y	Y	M	M	N	N	N	N	N	N	N	N	N	N	N	Y	N	N	N	N	N
dec	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	N	N	N	N	N	Y	N	N	N	N	N
int	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	N	N	N	N	N	Y	N	N	N	N	N
dur	Y	Y	N	N	N	N	Y	Y	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N
yMD	Y	Y	N	N	N	N	Y	Y	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N
dTD	Y	Y	N	N	N	N	Y	Y	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N
dT	Y	Y	N	N	N	N	N	N	N	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N
tim	Y	Y	N	N	N	N	N	N	N	N	Y	N	N	N	N	N	N	N	N	N	N	N	N
dat	Y	Y	N	N	N	N	N	N	N	Y	N	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N
gYM	Y	Y	N	N	N	N	N	N	N	N	N	N	Y	N	N	N	N	N	N	N	N	N	N
gYr	Y	Y	N	N	N	N	N	N	N	N	N	N	N	Y	N	N	N	N	N	N	N	N	N
gMD	Y	Y	N	N	N	N	N	N	N	N	N	N	N	N	Y	N	N	N	N	N	N	N	N
gDay	Y	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	Y	N	N	N	N	N	N	N

S\T	uA	str	flt	dbl	dec	int	dur	yMD	dTD	dT	tim	dat	gYM	gYr	gMD	gDay	gMon	bool	b64	hxB	aURI	QN	NOT
gMon	Y	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	Y	N	N	N	N	N	N
bool	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	N	N	N	N	N	Y	N	N	N	N	N
b64	Y	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	Y	Y	N	N	N
hxB	Y	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	Y	Y	N	N	N
aURI	Y	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	Y	N	N
QN	Y	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	Y	N
NOT	Y	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	M

The following sub-sections define the semantics of casting from a primitive type to a primitive type. Semantics of casting to and from a derived type are defined in sections **17.2 Casting to derived types**, **17.3 Casting from derived types to parent types**, **17.4 Casting within a branch of the type hierarchy** and **17.5 Casting across the type hierarchy**.

17.1.1 Casting from xs:string and xs:untypedAtomic

When the supplied value is an instance of `xs:string` or an instance of `xs:untypedAtomic`, it is treated as being a string value and mapped to a typed value of the target type as defined in [XML Schema Part 2: Datatypes Second Edition]. Whitespace normalization is applied as indicated by the whiteSpace facet for the datatype. The resulting whitespace-normalized string must be a valid lexical form for the datatype. The semantics of casting are identical to XML Schema validation. For example, `"13" cast as xs:unsignedInt` returns the `xs:unsignedInt` typed value `13`. This could also be written `xs:unsignedInt("13")`.

When casting from `xs:string` or `xs:untypedAtomic` to a derived type where the derived type is restricted by a pattern facet, the lexical form is first checked against the pattern before further casting is attempted (See **17.2 Casting to derived types**). If the lexical form does not conform to the pattern, error [err:FORG0001] is raised.

Consider a user-defined Schema whose target namespace is bound to the prefix `mySchema` which defines a restriction of `xs:boolean` called `trueBool` which allows only the lexical forms "`1`" and "`0`". `"true" cast as mySchema:trueBool` would fail with [err:FORG0001]. If the Schema also defines a datatype called `height` as a restriction of `xs:integer` with a maximum value of `84` then `"100" cast as mySchema:height` would also fail with [err:FORG0001].

Casting is permitted from `xs:string` and `xs:untypedAtomic` to any primitive atomic type or any atomic type derived by restriction, except `xs:QName` or `xs:NOTATION`. Casting to `xs:NOTATION` is not permitted because it is an abstract type.

Casting is permitted from `xs:string` literals to `xs:QName` and types derived from `xs:NOTATION`. If the argument to such a cast is computed dynamically, [err:XPTY0004]XP is raised if the value is of any type other than `xs:QName` or `xs:NOTATION` respectively (including the case where it is an `xs:string`). The process is described in more detail in **5.3 Constructor Functions for xs:QName and xs:NOTATION**.

In casting to numerics, if the value is too large or too small to be accurately represented by the implementation, it is handled as an overflow or underflow as defined in **6.2 Operators on Numeric Values**.

In casting to `xs:decimal` or to a type derived from `xs:decimal`, if the value is not too large or too small but nevertheless cannot be represented accurately with the number of decimal digits available to the implementation, the implementation may round to the nearest representable value or may raise a dynamic error [err:FOCA0006]. The choice of rounding algorithm and the choice between rounding and error behavior and is implementation-defined.

In casting to `xs:date`, `xs:dateTime`, `xs:gYear`, or `xs:gYearMonth` (or types derived from these), if the value is too large or too small to be represented by the implementation, error [err:FODT0001] is raised.

In casting to a duration value, if the value is too large or too small to be represented by the implementation, error [err:FODT0002] is raised.

For `xs:anyURI`, the extent to which an implementation validates the lexical form of `xs:anyURI` is ·implementation dependent·.

If the cast fails for any other reason, error [err:FORG0001] is raised.

17.1.2 Casting to xs:string and xs:untypedAtomic

Casting is permitted from any primitive type to the primitive types `xs:string` and `xs:untypedAtomic`.

When a value of any simple type is cast as `xs:string`, the derivation of the `xs:string` value *TV* depends on the *ST* and on the *SV*, as follows.

- If *ST* is `xs:string` or a type derived from `xs:string`, *TV* is *SV*.
- If *ST* is `xs:anyURI`, the type conversion is performed without escaping any characters.
- If *ST* is `xs:QName` or `xs:NOTATION`:
    - if the qualified name has a prefix, then *TV* is the concatenation of the prefix of *SV*, a single colon (:), and the local name of *SV*.
    - otherwise *TV* is the local-name.
- If *ST* is a numeric type, the following rules apply:
    - If *ST* is `xs:integer`, *TV* is the canonical lexical representation of *SV* as defined in [XML Schema Part 2: Datatypes Second Edition]. There is no decimal point.
    - If *ST* is `xs:decimal`, then:
        - If *SV* is in the value space of `xs:integer`, that is, if there are no significant digits after the decimal point, then the value is converted from an `xs:decimal` to an `xs:integer` and the resulting `xs:integer` is converted to an `xs:string` using the rule above.
        - Otherwise, the canonical lexical representation of *SV* is returned, as defined in [XML Schema Part 2: Datatypes Second Edition].
    - If *ST* is `xs:float` or `xs:double`, then:
        - *TV* will be an `xs:string` in the lexical space of `xs:double` or `xs:float` that when converted to an `xs:double` or `xs:float` under the rules of **17.1.1 Casting from xs:string and xs:untypedAtomic** produces a value that is equal to *SV*, or is "NaN" if *SV* is `NaN`. In addition, *TV* must satisfy the constraints in the following sub-bullets.
            - If *SV* has an absolute value that is greater than or equal to 0.000001 (one millionth) and less than 1000000 (one million), then the value is converted to an `xs:decimal` and the resulting `xs:decimal` is converted to an `xs:string` according to the rules above, as though using an implementation of `xs:decimal` that imposes no limits on the `totalDigits` or `fractionDigits` facets.
            - If *SV* has the value positive or negative zero, *TV* is "0" or "-0" respectively.
            - If *SV* is positive or negative infinity, *TV* is the string "`INF`" or "`-INF`" respectively.
            - In other cases, the result consists of a mantissa, which has the lexical form of an `xs:decimal`, followed by the letter "E", followed by an exponent which has the lexical form of an `xs:integer`. Leading zeroes and "+" signs are prohibited in the exponent. For the mantissa, there must be a decimal point, and there must be exactly one digit before the decimal point, which must be non-zero. The "+" sign is prohibited. There must be at least one digit after

the decimal point. Apart from this mandatory digit, trailing zero digits are prohibited.

Note:

The above rules allow more than one representation of the same value. For example, the `xs:float` value whose exact decimal representation is 1.26743223E15 might be represented by any of the strings "1.26743223E15", "1.26743222E15" or "1.26743224E15" (inter alia). It is implementation-dependent which of these representations is chosen.

- If *ST* is `xs:dateTime`, `xs:date` or `xs:time`, *TV* is the local value. The components of *TV* are individually cast to `xs:string` using the functions described in [casting-to-datetimes] and the results are concatenated together. The `year` component is cast to `xs:string` using `eg:convertYearToString`. The `month`, `day`, `hour` and `minute` components are cast to `xs:string` using `eg:convertTo2CharString`. The `second` component is cast to `xs:string` using `eg:convertSecondsToString`. The timezone component, if present, is cast to `xs:string` using `eg:convertTZtoString`.

  Note that the hours component of the resulting string will never be `"24"`. Midnight is always represented as `"00:00:00"`.
- If *ST* is `xs:yearMonthDuration` or `xs:dayTimeDuration`, *TV* is the canonical representation of *SV* as defined in **10.3.1 xs:yearMonthDuration** or **10.3.2 xs:dayTimeDuration**, respectively.
- If *ST* is `xs:duration` then let *SYM* be `SV cast as xs:yearMonthDuration`, and let *SDT* be `SV cast as xs:dayTimeDuration`; Now, let the next intermediate value, *TYM*, be `SYM cast as TT`, and let *TDT* be `SDT cast as TT`. If *TYM* is "P0M", then *TV* is *TDT*. Otherwise, *TYM* and *TDT* are merged according to the following rules:
  1. If *TDT* is "PT0S", then *TV* is *TYM*.
  2. Otherwise, *TV* is the concatenation of all the characters in *TYM* and all the characters except the first "P" and the optional negative sign in *TDT*.
- In all other cases, *TV* is the [XML Schema Part 2: Datatypes Second Edition] canonical representation of *SV*. For datatypes that do not have a canonical lexical representation defined an ·implementation dependent· canonical representation may be used.

To cast as `xs:untypedAtomic` the value is cast as `xs:string`, as described above, and the type annotation changed to `xs:untypedAtomic`.

Note:

The string representations of numeric values are backwards compatible with XPath 1.0 except for the special values positive and negative infinity, negative zero and values outside the range `1.0e-6` to `1.0e+6`.

17.1.3 Casting to numeric types

17.1.3.1 Casting to xs:float

When a value of any simple type is cast as `xs:float`, the `xs:float` *TV* is derived from the *ST* and the *SV* as follows:

- If *ST* is `xs:float`, then *TV* is *SV* and the conversion is complete.
- If *ST* is `xs:double`, then *TV* is obtained as follows:
  - if *SV* is the `xs:double` value `INF`, `-INF`, `NaN`, positive zero, or negative zero, then *TV* is the `xs:float` value `INF`, `-INF`, `NaN`, positive zero, or negative zero respectively.
  - otherwise, *SV* can be expressed in the form `m × 2^e` where the mantissa `m` and exponent `e` are signed `xs:integer`s whose value range is defined in [XML Schema Part 2: Datatypes Second Edition], and the following rules apply:

        - if `m` (the mantissa of *SV*) is outside the permitted range for the mantissa of an `xs:float` value `(-2^24-1 to +2^24-1)`, then it is divided by `2^N` where `N` is the lowest positive `xs:integer` that brings the result of the division within the permitted range, and the exponent `e` is increased by `N`. This is integer division (in effect, the binary value of the mantissa is truncated on the right). Let `M` be the mantissa and `E` the exponent after this adjustment.
        - if `E` exceeds `104` (the maximum exponent value in the value space of `xs:float`) then *TV* is the `xs:float` value `INF` or `-INF` depending on the sign of `M`.
        - if `E` is less than `-149` (the minimum exponent value in the value space of `xs:float`) then *TV* is the `xs:float` value positive or negative zero depending on the sign of `M`
        - otherwise, *TV* is the `xs:float` value `M × 2^E`.
- If *ST* is `xs:decimal`, or `xs:integer`, then *TV* is `xs:float(` *SV* `cast as xs:string)` and the conversion is complete.
- If *ST* is `xs:boolean`, *SV* is converted to `1.0E0` if *SV* is `true` and to `0.0E0` if *SV* is `false` and the conversion is complete.
- If *ST* is `xs:untypedAtomic` or `xs:string`, see **17.1.1 Casting from xs:string and xs:untypedAtomic**.

    Note:

    Implementations ·may· return negative zero for `"-0.0E0" cast as xs:float`. [XML Schema Part 2: Datatypes Second Edition] does not distinguish between the values positive zero and negative zero.

17.1.3.2 Casting to xs:double

When a value of any simple type is cast as `xs:double`, the `xs:double` value *TV* is derived from the *ST* and the *SV* as follows:

- If *ST* is `xs:double`, then *TV* is *SV* and the conversion is complete.
- If *ST* is `xs:float` or a type derived from `xs:float`, then *TV* is obtained as follows:
    - if *SV* is the `xs:float` value `INF`, `-INF`, `NaN`, positive zero, or negative zero, then *TV* is the `xs:double` value `INF`, `-INF`, `NaN`, positive zero, or negative zero respectively.
    - otherwise, *SV* can be expressed in the form `m × 2^e` where the mantissa `m` and exponent `e` are signed `xs:integer` values whose value range is defined in [XML Schema Part 2: Datatypes Second Edition], and *TV* is the `xs:double` value `m × 2^e`.
- If *ST* is `xs:decimal` or `xs:integer`, then *TV* is `xs:double(` *SV* `cast as xs:string)` and the conversion is complete.
- If *ST* is `xs:boolean`, *SV* is converted to `1.0E0` if *SV* is `true` and to `0.0E0` if *SV* is `false` and the conversion is complete.
- If *ST* is `xs:untypedAtomic` or `xs:string`, see **17.1.1 Casting from xs:string and xs:untypedAtomic**.

    Note:

    Implementations ·may· return negative zero for `"-0.0E0" cast as xs:double`. [XML Schema Part 2: Datatypes Second Edition] does not distinguish between the values positive zero and negative zero.

17.1.3.3 Casting to xs:decimal

When a value of any simple type is cast as `xs:decimal`, the `xs:decimal` value *TV* is derived from *ST* and *SV* as follows:

- If *ST* is `xs:decimal`, `xs:integer` or a type derived from them, then *TV* is *SV*, converted to an `xs:decimal` value if need be, and the conversion is complete.
- If *ST* is `xs:float` or `xs:double`, then *TV* is the `xs:decimal` value, within the set of `xs:decimal` values that the implementation is capable of representing, that is numerically closest to *SV*. If two values are equally close, then the one that is closest to zero is chosen. If *SV* is too large to be accommodated as an `xs:decimal`, (see [XML Schema Part 2: Datatypes Second Edition] for ·implementation-defined· limits on numeric values) an error is raised [err:FOCA0001]. If *SV* is one of the special `xs:float` or `xs:double` values `NaN`, `INF`, or `-INF`, an error is raised [err:FOCA0002].
- If *ST* is `xs:boolean`, *SV* is converted to `1.0` if *SV* is `1` or `true` and to `0.0` if *SV* is `0` or `false` and the conversion is complete.
- If *ST* is `xs:untypedAtomic` or `xs:string`, see **17.1.1 Casting from xs:string and xs:untypedAtomic**.

17.1.3.4 Casting to xs:integer

When a value of any simple type is cast as `xs:integer`, the `xs:integer` value *TV* is derived from *ST* and *SV* as follows:

- If *ST* is `xs:integer`, or a type derived from `xs:integer`, then *TV* is *SV*, converted to an `xs:integer` value if need be, and the conversion is complete.
- If *ST* is `xs:decimal`, `xs:float` or `xs:double`, then *TV* is *SV* with the fractional part discarded and the value converted to `xs:integer`. Thus, casting `3.1456` returns `3` and `-17.89` returns `-17`. Casting `3.124E1` returns `31`. If *SV* is too large to be accommodated as an integer, (see [XML Schema Part 2: Datatypes Second Edition] for ·implementation-defined· limits on numeric values) an error is raised [err:FOCA0003]. If *SV* is one of the special `xs:float` or `xs:double` values `NaN`, `INF`, or `-INF`, an error is raised [err:FOCA0002].
- If *ST* is `xs:boolean`, *SV* is converted to `1` if *SV* is `1` or `true` and to `0` if *SV* is `0` or `false` and the conversion is complete.
- If *ST* is `xs:untypedAtomic` or `xs:string`, see **17.1.1 Casting from xs:string and xs:untypedAtomic**.

17.1.4 Casting to duration types

When a value of type `xs:untypedAtomic`, `xs:string`, a type derived from `xs:string`, `xs:yearMonthDuration` or `xs:dayTimeDuration` is cast as `xs:duration`, `xs:yearMonthDuration` or `xs:dayTimeDuration`, *TV* is derived from *ST* and *SV* as follows:

- If *ST* is the same as *TT*, then *TV* is *SV*.
- If *ST* is `xs:duration`, or a type derived from `xs:duration`, but not `xs:dayTimeDuration` or a type derived from `xs:dayTimeDuration`, and *TT* is `xs:yearMonthDuration`, then *TV* is derived from *SV* by removing the day, hour, minute and second components from *SV*.
- If *ST* is `xs:duration`, or a type derived from `duration`, but not `xs:yearMonthDuration` or a type derived from `xs:yearMonthDuration`, and *TT* is `xs:dayTimeDuration`, then *TV* is derived from *SV* by removing the year and month components from *SV*.
- If *ST* is `xs:yearMonthDuration` or `xs:dayTimeDuration`, and *TT* is `xs:duration`, then *TV* is derived from *SV* as discussed in **17.3 Casting from derived types to parent types**.
- If *ST* is `xs:yearMonthDuration` and *TT* is `xs:dayTimeDuration`, the cast is permitted and returns a `xs:dayTimeDuration` with value 0.0 seconds.

- If *ST* is `xs:dayTimeDuration` and *TT* is `xs:yearMonthDuration`, the cast is permitted and returns a `xs:yearMonthDuration` with value 0 months.
- If *ST* is `xs:untypedAtomic` or `xs:string`, see **17.1.1 Casting from xs:string and xs:untypedAtomic**.

Note that casting from `xs:duration` to `xs:yearMonthDuration` or `xs:dayTimeDuration` loses information. To avoid this, users can cast the `xs:duration` value to both an `xs:yearMonthDuration` and an `xs:dayTimeDuration` and work with both values.

17.1.5 Casting to date and time types

In several situations, casting to date and time types requires the extraction of a component from *SV* or from the result of `fn:current-dateTime` and converting it to an `xs:string`. These conversions must follow certain rules. For example, converting an `xs:integer` year value requires converting to an `xs:string` with four or more characters, preceded by a minus sign if the value is negative.

This document defines four functions to perform these conversions. These functions are for illustrative purposes only and make no recommendations as to style or efficiency. References to these functions from the following text are not normative.

The arguments to these functions come from functions defined in this document. Thus, the functions below assume that they are correct and do no range checking on them.

```
declare function eg:convertYearToString($year as xs:integer) as xs:string
{
   let $plusMinus := if ($year >= 0) then "" else "-"
   let $yearString := fn:abs($year) cast as xs:string
   let $length := fn:string-length($yearString)
   return
     if ($length = 1)  then fn:concat($plusMinus, "000", $yearString)
     else
     if ($length = 2)  then fn:concat($plusMinus, "00", $yearString)
       else
       if ($length = 3)  then fn:concat($plusMinus, "0", $yearString)
       else fn:concat($plusMinus, $yearString)
}
```

```
declare function eg:convertTo2CharString($value as xs:integer) as xs:string
{
   let $string := $value cast as xs:string
   return
     if (fn:string-length($string) = 1) then fn:concat("0", $string)
     else $string
}
```

```
declare function eg:convertSecondsToString($seconds as xs:decimal) as xs:string
{
   let $string := $seconds cast as xs:string
   let $intLength := fn:string-length(($seconds cast as xs:integer) cast as xs:string)
   return
     if ($intLength = 1) then fn:concat("0", $string)
     else $string
}
```

```
declare function eg:convertTZtoString($tz as xs:dayTimeDuration?) as xs:string
{
   if (empty($tz))
     then ""
   else if ($tz eq xs:dayTimeDuration('PT0S'))
```

```
      then "Z"
    else
      let $tzh := fn:hours-from-duration($tz)
      let $tzm := fn:minutes-from-duration($tz)
      let $plusMinus := if ($tzh >= 0) then "+" else "-"
      let $tzhString := eg:convertTo2CharString(fn:abs($tzh))
      let $tzmString := eg:convertTo2CharString(fn:abs($tzm))
      return fn:concat($plusMinus, $tzhString, ":", $tzmString)
}
```

Conversion from primitive types to date and time types follows the rules below.

1. When a value of any primitive type is cast as `xs:dateTime`, the `xs:dateTime` value *TV* is derived from *ST* and *SV* as follows:
   - If *ST* is `xs:dateTime`, then *TV* is *SV*.
   - If *ST* is `xs:date`, then let *SYR* be `eg:convertYearToString( fn:year-from-date(` *SV* `))`, let *SMO* be `eg:convertTo2CharString( fn:month-from-date(` *SV* `))`, let *SDA* be `eg:convertTo2CharString( fn:day-from-date(` *SV* `))` and let *STZ* be `eg:convertTZtoString( fn:timezone-from-date(` *SV* `))`; *TV* is `xs:dateTime( fn:concat(` *SYR* `, '-',` *SMO* `, '-',` *SDA* `, 'T00:00:00 ',` *STZ* `) )`.
   - If *ST* is `xs:untypedAtomic` or `xs:string`, see **17.1.1 Casting from xs:string and xs:untypedAtomic**.
2. When a value of any primitive type is cast as `xs:time`, the `xs:time` value *TV* is derived from *ST* and *SV* as follows:
   - If *ST* is `xs:time`, then *TV* is *SV*.
   - If *ST* is `xs:dateTime`, then *TV* is `xs:time( fn:concat( eg:convertTo2CharString( fn:hours-from-dateTime(` *SV* `)), ':', eg:convertTo2CharString( fn:minutes-from-dateTime(` *SV* `)), ':', eg:convertSecondsToString( fn:seconds-from-dateTime(` *SV* `)), eg:convertTZtoString( fn:timezone-from-dateTime(` *SV* `)) ))`.
   - If *ST* is `xs:untypedAtomic` or `xs:string`, see **17.1.1 Casting from xs:string and xs:untypedAtomic**.
3. When a value of any primitive type is cast as `xs:date`, the `xs:date` value *TV* is derived from *ST* and *SV* as follows:
   - If *ST* is `xs:date`, then *TV* is *SV*.
   - If *ST* is `xs:dateTime`, then let *SYR* be `eg:convertYearToString( fn:year-from-dateTime(` *SV* `))`, let *SMO* be `eg:convertTo2CharString( fn:month-from-dateTime(` *SV* `))`, let *SDA* be `eg:convertTo2CharString( fn:day-from-dateTime(` *SV* `))` and let *STZ* be `eg:convertTZtoString(fn:timezone-from-dateTime(` *SV* `))`; *TV* is `xs:date( fn:concat(` *SYR* `, '-',` *SMO* `, '-',` *SDA*, *STZ* `) )`.
   - If *ST* is `xs:untypedAtomic` or `xs:string`, see **17.1.1 Casting from xs:string and xs:untypedAtomic**.
4. When a value of any primitive type is cast as `xs:gYearMonth`, the `xs:gYearMonth` value *TV* is derived from *ST* and *SV* as follows:
   - If *ST* is `xs:gYearMonth`, then *TV* is *SV*.
   - If *ST* is `xs:dateTime`, then let *SYR* be `eg:convertYearToString( fn:year-from-dateTime(` *SV* `))`, let *SMO* be `eg:convertTo2CharString( fn:month-from-dateTime(` *SV* `))` and let *STZ* be `eg:convertTZtoString( fn:timezone-from-dateTime(` *SV* `))`; *TV* is `xs:gYearMonth( fn:concat(` *SYR* `, '-',` *SMO*, *STZ* `) )`.
   - If *ST* is `xs:date`, then let *SYR* be `eg:convertYearToString( fn:year-from-date(` *SV* `))`, let *SMO* be `eg:convertTo2CharString( fn:month-from-date(` *SV* `))` and let *STZ* be `eg:convertTZtoString( fn:timezone-from-date(` *SV* `))`; *TV* is `xs:gYearMonth( fn:concat(` *SYR* `, '-',` *SMO*, *STZ* `) )`.

- If *ST* is `xs:untypedAtomic` or `xs:string`, see **17.1.1 Casting from xs:string and xs:untypedAtomic**.

5. When a value of any primitive type is cast as `xs:gYear`, the `xs:gYear` value *TV* is derived from *ST* and *SV* as follows:
   - If *ST* is `xs:gYear`, then *TV* is *SV*.
   - If *ST* is `xs:dateTime`, let *SYR* be `eg:convertYearToString( fn:year-from-dateTime(` *SV* `))` and let *STZ* be `eg:convertTZtoString( fn:timezone-from-dateTime(` *SV* `))`; *TV* is `xs:gYear(fn:concat(` *SYR*, *STZ* `))`.
   - If *ST* is `xs:date`, let *SYR* be `eg:convertYearToString( fn:year-from-date(` *SV* `))`; and let *STZ* be `eg:convertTZtoString( fn:timezone-from-date(` *SV* `))`; *TV* is `xs:gYear(fn:concat(` *SYR*, *STZ* `))`.
   - If *ST* is `xs:untypedAtomic` or `xs:string`, see **17.1.1 Casting from xs:string and xs:untypedAtomic**.
6. When a value of any primitive type is cast as `xs:gMonthDay`, the `xs:gMonthDay` value *TV* is derived from *ST* and *SV* as follows:
   - If *ST* is `xs:gMonthDay`, then *TV* is *SV*.
   - If *ST* is `xs:dateTime`, then let *SMO* be `eg:convertTo2CharString( fn:month-from-dateTime(` *SV* `))`, let *SDA* be `eg:convertTo2CharString( fn:day-from-dateTime(` *SV* `))` and let *STZ* be `eg:convertTZtoString( fn:timezone-from-dateTime(` *SV* `))`; *TV* is `xs:gYearMonth( fn:concat( '--',` *SMO* `'-',` *SDA*, *STZ* `) )`.
   - If *ST* is `xs:date`, then let *SMO* be `eg:convertTo2CharString( fn:month-from-date(` *SV* `))`, let *SDA* be `eg:convertTo2CharString( fn:day-from-date(` *SV* `))` and let *STZ* be `eg:convertTZtoString( fn:timezone-from-date(` *SV* `))`; *TV* is `xs:gYearMonth( fn:concat( '--',` *SMO* `, '-',` *SDA*, *STZ* `) )`.
   - If *ST* is `xs:untypedAtomic` or `xs:string`, see **17.1.1 Casting from xs:string and xs:untypedAtomic**.
7. When a value of any primitive type is cast as `xs:gDay`, the `xs:gDay` value *TV* is derived from *ST* and *SV* as follows:
   - If *ST* is `xs:gDay`, then *TV* is *SV*.
   - If *ST* is `xs:dateTime`, then let *SDA* be `eg:convertTo2CharString( fn:day-from-dateTime(` *SV* `))` and let *STZ* be `eg:convertTZtoString( fn:timezone-from-dateTime(` *SV* `))`; *TV* is `xs:gDay( fn:concat( '---',` *SDA*, *STZ* `))`.
   - If *ST* is `xs:date`, then let *SDA* be `eg:convertTo2CharString( fn:day-from-date(` *SV* `))` and let *STZ* be `eg:convertTZtoString( fn:timezone-from-date(` *SV* `))`; *TV* is `xs:gDay( fn:concat( '---',` *SDA*, *STZ* `))`.
   - If *ST* is `xs:untypedAtomic` or `xs:string`, see **17.1.1 Casting from xs:string and xs:untypedAtomic**.
8. When a value of any primitive type is cast as `xs:gMonth`, the `xs:gMonth` value *TV* is derived from *ST* and *SV* as follows:
   - If *ST* is `xs:gMonth`, then *TV* is *SV*.
   - If *ST* is `xs:dateTime`, then let *SMO* be `eg:convertTo2CharString( fn:month-from-dateTime(` *SV* `))` and let *STZ* be `eg:convertTZtoString( fn:timezone-from-dateTime(` *SV* `))`; *TV* is `xs:gMonth( fn:concat( '--' ,` *SMO*, *STZ* `))`.
   - If *ST* is `xs:date`, then let *SMO* be `eg:convertTo2CharString( fn:month-from-date(` *SV* `))` and let *STZ* be `eg:convertTZtoString( fn:timezone-from-date(` *SV* `))`; *TV* is `xs:gMonth( fn:concat( '--',` *SMO*, *STZ* `))`.
   - If *ST* is `xs:untypedAtomic` or `xs:string`, see **17.1.1 Casting from xs:string and xs:untypedAtomic**.

17.1.6 Casting to xs:boolean

When a value of any primitive type is cast as `xs:boolean`, the `xs:boolean` value *TV* is derived from *ST* and *SV* as follows:

- If *ST* is `xs:boolean`, then *TV* is *SV*.
- If *ST* is `xs:float`, `xs:double`, `xs:decimal` or `xs:integer` and *SV* is `0`, `+0`, `-0`, `0.0`, `0.0E0` or `NaN`, then *TV* is `false`.
- If *ST* is `xs:float`, `xs:double`, `xs:decimal` or `xs:integer` and *SV* is not one of the above values, then *TV* is `true`.
- If *ST* is `xs:untypedAtomic` or `xs:string`, see **17.1.1 Casting from xs:string and xs:untypedAtomic**.

17.1.7 Casting to xs:base64Binary and xs:hexBinary

Values of type `xs:base64Binary` can be cast as `xs:hexBinary` and vice versa, since the two types have the same value space. Casting to `xs:base64Binary` and `xs:hexBinary` is also supported from the same type and from `xs:untypedAtomic`, `xs:string` and subtypes of `xs:string` using [XML Schema Part 2: Datatypes Second Edition] semantics.

17.1.8 Casting to xs:anyURI

Casting to `xs:anyURI` is supported only from the same type, `xs:untypedAtomic` or `xs:string`.

When a value of any primitive type is cast as `xs:anyURI`, the `xs:anyURI` value *TV* is derived from the *ST* and *SV* as follows:

- If *ST* is `xs:untypedAtomic` or `xs:string` see **17.1.1 Casting from xs:string and xs:untypedAtomic**.

17.2 Casting to derived types

Casting a value to a derived type can be separated into four cases. Note that `xs:untypedAtomic`, `xs:integer` and the two derived types of `xs:duration`:`xs:yearMonthDuration` and `xs:dayTimeDuration` are treated as primitive types.

1. When *SV* is an instance of a type that is derived by restriction from *TT*. This is described in section **17.3 Casting from derived types to parent types**.
2. When *SV* is an instance of a type derived by restriction from the same primitive type as *TT*. This is described in **17.4 Casting within a branch of the type hierarchy**.
3. When the derived type is derived, directly or indirectly, from a different primitive type than the primitive type of *ST*. This is described in **17.5 Casting across the type hierarchy**.
4. When *SV* is an instance of the *TT*, the cast always succeeds (Identity cast).

17.3 Casting from derived types to parent types

Except in the case of `xs:NOTATION`, it is always possible to cast a value of any atomic type to an atomic type from which it is derived, directly or indirectly, by restriction. For example, it is possible to cast an `xs:unsignedShort` to an `xs:unsignedInt`, an `xs:integer`, or an `xs:decimal`. Since the value space of the original type is a subset of the value space of the target type, such a cast is always successful. The result will have the same value as the original, but will have a new type annotation.

17.4 Casting within a branch of the type hierarchy

It is possible to cast an *SV* to a *TT* if the type of the *SV* and the *TT* type are both derived by restriction (directly or indirectly) from the same primitive type, provided that the supplied value conforms to the constraints implied by the facets of the target type. This includes the case where the target type is derived from the type of the supplied value, as well as the case where the type of the supplied value is derived from the target type. For example, an instance of `xs:byte` can be cast as `xs:unsignedShort`, provided the value is not negative.

If the value does not conform to the facets defined for the target type, then an error is raised [err:FORG0001]. See [XML Schema Part 2: Datatypes Second Edition]. In the case of the pattern facet (which applies to the lexical space rather than the value space), the pattern is tested against the canonical lexical representation of the value, as defined for the source type (or the result of casting the value to an `xs:string`, in the case of types that have no canonical lexical representation defined for them).

Note that this will cause casts to fail if the pattern excludes the canonical lexical representation of the source type. For example, if the type `my:distance` is defined as a restriction of `xs:decimal` with a pattern that requires two digits after the decimal point, casting of an `xs:integer` to `my:distance` will always fail, because the canonical representation of an `xs:integer` does not conform to this pattern.

In some cases, casting from a parent type to a derived type requires special rules. See **17.1.4 Casting to duration types** for rules regarding casting to `xs:yearMonthDuration` and `xs:dayTimeDuration`. See **17.4.1 Casting to xs:ENTITY**, below, for casting to `xs:ENTITY` and types derived from it.

17.4.1 Casting to xs:ENTITY

[XML Schema Part 2: Datatypes Second Edition] says that "The value space of ENTITY is the set of all strings that match the NCName production ... and have been declared as an unparsed entity in a document type definition." However, [XSL Transformations (XSLT) Version 2.0] and [XQuery 1.0: An XML Query Language] do not check that constructed values of type `xs:ENTITY` match declared unparsed entities. Thus, this rule is relaxed in this specification and, in casting to `xs:ENTITY` and types derived from it, no check is made that the values correspond to declared unparsed entities.

17.5 Casting across the type hierarchy

When the *ST* and the *TT* are derived, directly or indirectly, from different primitive types, this is called casting across the type hierarchy. Casting across the type hierarchy is logically equivalent to three separate steps performed in order. Errors can occur in either of the latter two steps.

1. Cast the *SV*, up the hierarchy, to the primitive type of the source, as described in **17.3 Casting from derived types to parent types**.
    a. If *SV* is an instance of `xs:string` or `xs:untypedAtomic`, check its value against the pattern facet of *TT*, and raise an error [err:FORG0001] if the check fails.
2. Cast the value to the primitive type of *TT*, as described in **17.1 Casting from primitive types to primitive types**.
    - If *TT* is derived from `xs:NOTATION`, assume for the purposes of this rule that casting to `xs:NOTATION` succeeds.
3. Cast the value down to the *TT*, as described in **17.4 Casting within a branch of the type hierarchy**

A References

A.1 Normative References

IEEE 754-1985
IEEE. *IEEE Standard for Binary Floating-Point Arithmetic.*

Locale Data Markup Language
Unicode Technical Standard #35, Locale Data Markup Language. Available at: http://www.unicode.org/reports/tr35/

RFC 2396
IETF. *RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax.* Available at: http://www.ietf.org/rfc/rfc2396.txt

RFC 3986
IETF. *RFC 3986: Uniform Resource Identifiers (URI): Generic Syntax.* Available at: http://www.ietf.org/rfc/rfc3986.txt

RFC 3987
IETF. *RFC 3987: Internationalized Resource Identifiers (IRIs).* Available at: http://www.ietf.org/rfc/rfc3987.txt

Character Model for the World Wide Web 1.0: Fundamentals
Character Model for the World Wide Web 1.0: Fundamentals. Available at: http://www.w3.org/TR/2005/REC-charmod-20050215/

Character Model for the World Wide Web 1.0: Normalization
Character Model for the World Wide Web 1.0: Normalization, Last Call Working Draft. Available at: http://www.w3.org/TR/2005/WD-charmod-norm-20051027/

ISO 10967
ISO (International Organization for Standardization). *ISO/IEC 10967-1:1994, Information technology—Language Independent Arithmetic—Part 1: Integer and floating point arithmetic* [Geneva]: International Organization for Standardization, 1994. Available from: http://www.iso.org/

The Unicode Standard
The Unicode Consortium, Reading, MA, Addison-Wesley, 2003. *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; currently, Version 4.0.00, Addison-Wesley, 2003 ISBN 0-321-18578-1

Unicode Collation Algorithm
Unicode Technical Standard #10, Unicode Collation Algorithm. Available at: http://www.unicode.org/reports/tr10/

Unicode Regular Expressions
Unicode Technical Standard #18, Unicode Regular Expressions. Available at: http://www.unicode.org/reports/tr18/

Extensible Markup Language (XML) 1.0 Recommendation (Third Edition)
World Wide Web Consortium. *Extensible Markup Language (XML) 1.0 Third Edition.* Available at: http://www.w3.org/TR/REC-xml/

Extensible Markup Language (XML) 1.1 Recommendation
World Wide Web Consortium. *Extensible Markup Language (XML) 1.1.* Available at: http://www.w3.org/TR/2004/REC-xml11-20040204/

XML Path Language (XPath) 2.0
World Wide Web Consortium. XML Path Language (XPath) Version 2.0. Available at: http://www.w3.org/TR/xpath20/

XSL Transformations (XSLT) Version 2.0
World Wide Web Consortium. XSL Transformations Version 2.0. Available at: http://www.w3.org/TR/xslt20/

XQuery 1.0 and XPath 2.0 Data Model
World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Data Model (XDM). Available at: http://www.w3.org/TR/xpath-datamodel/

XQuery 1.0 and XPath 2.0 Formal Semantics
World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Formal Semantics. Available at: http://www.w3.org/TR/xquery-semantics/

XQuery 1.0: An XML Query Language

World Wide Web Consortium. XQuery 1.0: An XML Query Language. Available at: http://www.w3.org/TR/xquery/

XML Schema Part 1: Structures Second Edition
XML Schema Part 1: Structures Second Edition, Oct 28 2004. Available at: http://www.w3.org/TR/xmlschema-1/

XML Schema Part 2: Datatypes Second Edition
XML Schema Part 2: Datatypes Second Edition, Oct. 28 2004. Available at: http://www.w3.org/TR/xmlschema-2/

Namespaces in XML
Namespaces in XML. Available at: http://www.w3.org/TR/1999/REC-xml-names-19990114/

A.2 Non-normative References

HTML 4.0
HTML 4.01 Recommendation, 24 December 1999. Available at: http://www.w3.org/TR/REC-html40/

ISO 8601
ISO (International Organization for Standardization). *Representations of dates and times, 2000-08-03.* Available from: http://www.iso.org/"

Working With Timezones
World Wide Web Consortium Working Group Note. *Working With Timezones, October 13, 2005.* Available at: http://www.w3.org/TR/2005/NOTE-timezone-20051013/

XML Path Language (XPath) Version 1.0
World Wide Web Consortium. XML Path Language (XPath) Version 1.0 Available at: http://www.w3.org/TR/xpath/

B Error Summary

The error text provided with these errors is non-normative.

err:FOER0000, Unidentified error.

Unidentified error.

err:FOAR0001, Division by zero.

This error is raised whenever an attempt is made to divide by zero.

err:FOAR0002, Numeric operation overflow/underflow.

This error is raised whenever numeric operations result in an overflow or underflow.

err:FOCA0001, Input value too large for decimal.
err:FOCA0002, Invalid lexical value.
err:FOCA0003, Input value too large for integer.
err:FOCA0005, NaN supplied as float/double value.
err:FOCA0006, String to be cast to decimal has too many digits of precision.
err:FOCH0001, Code point not valid.
err:FOCH0002, Unsupported collation.
err:FOCH0003, Unsupported normalization form.
err:FOCH0004, Collation does not support collation units.
err:FODC0001, No context document.
err:FODC0002, Error retrieving resource.
err:FODC0003, Function stability not defined.
err:FODC0004, Invalid argument to fn:collection.
err:FODC0005, Invalid argument to fn:doc or fn:doc-available.
err:FODT0001, Overflow/underflow in date/time operation.

err:FODT0002, Overflow/underflow in duration operation.
err:FODT0003, Invalid timezone value.
err:FONS0004, No namespace found for prefix.
err:FONS0005, Base-uri not defined in the static context.
err:FORG0001, Invalid value for cast/constructor.
err:FORG0002, Invalid argument to fn:resolve-uri().
err:FORG0003, fn:zero-or-one called with a sequence containing more than one item.
err:FORG0004, fn:one-or-more called with a sequence containing no items.
err:FORG0005, fn:exactly-one called with a sequence containing zero or more than one item.
err:FORG0006, Invalid argument type.
err:FORG0008, The two arguments to fn:dateTime have inconsistent timezones.
err:FORG0009, Error in resolving a relative URI against a base URI in fn:resolve-uri.
err:FORX0001, Invalid regular expression flags.
err:FORX0002, Invalid regular expression.
err:FORX0003, Regular expression matches zero-length string.
err:FORX0004, Invalid replacement string.
err:FOTY0012, Argument node does not have a typed value.

C Compatibility with XPath 1.0 (Non-Normative)

This appendix summarizes the relationship between certain functions defined in [XML Path Language (XPath) Version 1.0] and the corresponding functions defined in this document. The first column of the table provides the signature of functions defined in this document. The second column provides the signature of the corresponding function in [XML Path Language (XPath) Version 1.0]. The third column discusses the differences in the semantics of the corresponding functions. The functions appear in the order they appear in [XML Path Language (XPath) Version 1.0].

The evaluation of the arguments to the functions defined in this document depends on whether the XPath 1.0 compatibility mode is on or off. See [XML Path Language (XPath) 2.0]. If the mode is on, the following conversions are applied, in order, before the argument value is passed to the function:

- If the expected type is a single item or an optional single item, (examples: `xs:string`, `xs:string?`, `xs:untypedAtomic`, `xs:untypedAtomic?`, `node()`, `node()?`, `item()`, `item()?`), then the given value `V` is effectively replaced by `fn:subsequence(V, 1, 1)`.
- If the expected type is `xs:string` or `xs:string?`, then the given value `V` is effectively replaced by `fn:string(V)`.
- If the expected type is numeric or optional numeric, then the given value `V` is effectively replaced by `fn:number(V)`.
- Otherwise, the given value is unchanged.

XQuery 1.0 and XPath 2.0	XPath 1.0	Notes
fn:last() as *xs:integer*	`last() => number`	Precision of numeric results may be different.
fn:position() as *xs:integer*	`position() => number`	Precision of numeric results may be different.
fn:count($arg as *item**) as *xs:integer*	`count(node-set) => number`	Precision of numeric results may be different.
fn:id($arg as *xs:string**) as *element()**	`id(object) => node-`	XPath 2.0 behavior is different for

	`set`	boolean and numeric arguments. The recognition of a node as an id value is sensitive to the manner in which the datamodel is constructed. In XPath 1.0 the whole string is treated as a unit. In XPath 2.0 each string is treated as a list.

`fn:local-name``()` `as` *`xs:string`*
`fn:local-name``($arg as` *`node()?`*`) as` *`xs:string`*
	`local-name(node-set?) => string`	If compatibility mode is off, an error will occur if argument has more than one node.

`fn:namespace-uri``()` `as` *`xs:string`*
`fn:namespace-uri``($arg as` *`node?`*`) as` *`xs:string`*
	`namespace-uri(node-set?) => string`	If compatibility mode is off, an error will occur if argument has more than one node.

`fn:name``($arg as` *`node()?`*`) as` *`xs:string`*	`name(node-set?) => string`	If compatibility mode is off, an error will occur if argument has more than one node. The rules for determining the prefix are more precisely defined in [XML Path Language (XPath) 2.0]. Function is not "well-defined" for parentless attribute nodes.

`fn:string``()` `as` *`xs:string`*
`fn:string``($arg as` *`item()?`*`) as` *`xs:string`*
	`string(object) => string`	If compatibility mode is off, an error will occur if argument has more than one node. Representations of numeric values are XPath 1.0 compatible except for the special values positive and negative infinity, and for values outside the range 1.0e-6 to 1.0e+6.

`fn:concat``($arg1 as` *`xs:anyAtomicType?`*`, $arg2 as` *`xs:anyAtomicType?`*`, ... ) as` *`xs:string`*	`concat(string, string, string*) => string`	If compatibility mode is off, an error will occur if an argument

		has more than one node. If compatibility mode on, the first node in the sequence is used.
fn:starts-with($arg1 as *xs:string?*, $arg2 as *xs:string?*) as *xs:boolean* **fn:starts-with**($arg1 as *xs:string?*, $arg2 as *xs:string?*, $collation as *xs:string*) as *xs:boolean*	`starts-with(string, string) => boolean`	If compatibility mode is off, an error will occur if either argument has more than one node or is a number or a boolean. If compatibility mode is on, implicit conversion is performed.
fn:contains($arg1 as *xs:string?*, $arg2 as *xs:string?*) as *xs:boolean* **fn:contains**($arg1 as *xs:string?*, $arg2 as *xs:string?*, $collation as *xs:string*) as *xs:boolean*	`contains(string, string) => boolean`	If compatibility mode is off, an error will occur if either argument has more than one node or is a number or a boolean. If compatibility mode is on, implicit conversion is performed.
fn:substring-before($arg1 as *xs:string?*, $arg2 as *xs:string?*) as *xs:string* **fn:substring-before**($arg1 as *xs:string?*, $arg2 as *xs:string?*, $collation as *xs:string*) as *xs:string*	`substring-before(string, string) => string`	If compatibility mode is off, an error will occur if either argument has more than one node or is a number or a boolean. If compatibility mode is on, implicit conversion is performed.
fn:substring-after($arg1 as *xs:string?*, $arg2 as *xs:string?*) as *xs:string* **fn:substring-after**($arg1 as *xs:string?*, $arg2 as *xs:string?*, $collation as *xs:string*) as *xs:string*	`substring-after(string, string) => string`	If compatibility mode is off, an error will occur if either argument has more than one node or is a number or a boolean. If compatibility mode is on, implicit conversion is performed.
fn:substring($sourceString as *xs:string?*, $startingLoc as *xs:double*) as *xs:string* **fn:substring**($sourceString as *xs:string?*,	`substring(string, number, number?) => string`	If compatibility mode is off, an error will occur if `$sourceString` has more than one node or is a number or a

`$startingLoc as xs:double,`
`$length as xs:double) as xs:string`
		boolean. If compatibility mode is on, implicit conversion is performed.

`fn:string-length``($arg as xs:string?) as xs:integer?`
`fn:string-length``() as xs:integer?`
	`string-length(string?) => number`	If compatibility mode is off, numbers and booleans will give errors for first arg. Also, multiple nodes will give error.

`fn:normalize-space``($arg as xs:string?) as xs:string`
`fn:normalize-space``() as xs:string`
	`normalize-space(string?) => string`	If compatibility mode is off, an error will occur if `$arg` has more than one node or is a number or a boolean. If compatibility mode is on, implicit conversion is performed.

`fn:translate``($arg as xs:string?, $mapString as xs:string, $transString as xs:string) as xs:string`	`translate(string, string, string)=> string`	.
`fn:boolean``($arg as item()*) as xs:boolean`	`boolean(object) => boolean`	
`fn:not``($arg as item()*) as xs:boolean`	`not(boolean) => boolean`	
`fn:true``() as xs:boolean`	`true() => boolean`	
`fn:false``() as xs:boolean`	`false() => boolean`	
`fn:lang``($testlang as xs:string) as xs:boolean`	`lang(string) => boolean`	If compatibility mode is off, numbers and booleans will give errors. Also, multiple nodes will give error. If compatibility mode is on, implicit conversion is performed.

`fn:number``() as xs:double`
`fn:number``($arg as xs:anyAtomicType?) as xs:double`
	`number(object?) => number`	Error if argument has more than one node when not in compatibility node.

`fn:sum``($arg as xs:anyAtomicType*) as` ***`xs:anyAtomicType`***	`sum(node-set) => number`	2.0 raises an error if sequence contains values that cannot be added together such as NMTOKENS and other subtypes of string. 1.0 returns `NaN`.
`fn:floor``($arg as numeric?) as` ***`numeric?`***	`floor(number)=>`	In 2.0, if argument is

	number	`()`, the result is `()`. In 1.0, the result is `NaN`. If compatibility mode is off, an error will occur with more than one node. If compatibility mode is on, implicit conversion is performed.
`fn:ceiling``($arg as` *`numeric?`*`) as` ***`numeric?`***	`ceiling(number)=> number`	In 2.0, if argument is `()`, the result is `()`. In 1.0, the result is `NaN`. If compatibility mode is off, an error will occur with more than one node. If compatibility mode is on, implicit conversion is performed.
`fn:round``($arg as` *`numeric?`*`) as` ***`numeric?`***	`round(number)=> number`	In 2.0, if argument is `()`, the result is `()`. In 1.0, the result is `NaN`. If compatibility mode is off, an error will occur with more than one node. If compatibility mode is on, implicit conversion is performed.

D Illustrative User-written Functions (Non-Normative)

Certain functions that were proposed for inclusion in this function library have been excluded on the basis that it is straightforward for users to implement these functions themselves using XSLT 2.0 or XQuery 1.0.

This Appendix provides sample implementations of some of these functions.

To emphasize that these functions are examples of functions that vendors may write, their names carry the prefix 'eg'. Vendors are free to define such functions in any namespace. A group of vendors may also choose to create a collection of such useful functions and put them in a common namespace.

D.1 eg:if-empty and eg:if-absent

In some situations, users may want to provide default values for missing information that may be signaled by elements that are omitted, have no value or have the empty sequence as their value. For example, a missing middle initial may be indicated by omitting the element or a non-existent bonus signaled with an empty sequence. This section includes examples of functions that provide such defaults. These functions return `xs:anyAtomicType*`. Users may want to write functions that return more specific types.

D.1.1 eg:if-empty

```
eg:if-empty($node  as node()?,
            $value as xs:anyAtomicType) as xs:anyAtomicType*
```

If the first argument is the empty sequence or an element without simple or complex content, if-empty() returns the second argument; otherwise, it returns the content of the first argument.

XSLT implementation

```
<xsl:function name="eg:if-empty" as="xs:anyAtomicType*">
  <xsl:param name="node" as="node()?"/>
  <xsl:param name="value" as="xs:anyAtomicType"/>
  <xsl:choose>
    <xsl:when test="$node and $node/child::node()">
      <xsl:sequence select="fn:data($node)"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:sequence select="$value"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:function>
```

XQuery implementation

```
declare function eg:if-empty (
  $node as node()?,
  $value as xs:anyAtomicType) as xs:anyAtomicType*
{
  if ($node and $node/child::node())
          then fn:data($node)
          else $value
}
```

D.1.2 eg:if-absent

```
eg:if-absent($node  as node()?,
             $value as xs:anyAtomicType) as xs:anyAtomicType*
```

If the first argument is the empty sequence, if-absent() returns the second argument; otherwise, it returns the content of the first argument.

XSLT implementation

```
<xsl:function name="eg:if-absent">
  <xsl:param name="node" as="node()?"/>
  <xsl:param name="value" as="xs:anyAtomicType"/>
  <xsl:choose>
    <xsl:when test="$node">
      <xsl:sequence select="fn:data($node)"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:sequence select="$value"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:function>
```

XQuery implementation

```
declare function eg:if-absent (
  $node as node()?,
  $value as xs:anyAtomicType) as xs:anyAtomicType*
{
  if ($node)
    then fn:data($node)
    else $value
}
```

D.2 union, intersect and except on sequences of values

D.2.1 eg:value-union

```
eg:value-union($arg1 as xs:anyAtomicType*,
               $arg2 as xs:anyAtomicType*) as xs:anyAtomicType*
```

This function returns a sequence containing all the distinct items in $arg1 and $arg2, in an undefined order.

XSLT implementation

```
xsl:function name="eg:value-union" as="xs:anyAtomicType*">
  <xsl:param name="arg1" as="xs:anyAtomicType*"/>
  <xsl:param name="arg2" as="xs:anyAtomicType*"/>
  <xsl:sequence
     select="fn:distinct-values(($arg1, $arg2))"/>
</xsl:function>
```

XQuery implementation

```
declare function eg:value-union (
  $arg1 as xs:anyAtomicType*,
  $arg2 as xs:anyAtomicType*) as xs:anyAtomicType*
{
  fn:distinct-values(($arg1, $arg2))
}
```

D.2.2 eg:value-intersect

```
eg:value-intersect($arg1 as xs:anyAtomicType*,
                   $arg2 as xs:anyAtomicType*) as xs:anyAtomicType*
```

This function returns a sequence containing all the distinct items that appear in both $arg1 and $arg2, in an undefined order.

XSLT implementation>

```
<xsl:function name="eg:value-intersect" as="xs:anyAtomicType*">
  <xsl:param name="arg1" as="xs:anyAtomicType*"/>
  <xsl:param name="arg2" as="xs:anyAtomicType*"/>
  <xsl:sequence
     select="fn:distinct-values($arg1[.=$arg2])"/>
</xsl:function>
```

XQuery implementation

```
declare function eg:value-intersect (
  $arg1 as xs:anyAtomicType*,
  $arg2 as xs:anyAtomicType* ) as xs:anyAtomicType*
{
  fn:distinct-values($arg1[.=$arg2])
}
```

D.2.3 eg:value-except

```
eg:value-except($arg1 as xs:anyAtomicType*,
                $arg2 as xs:anyAtomicType*) as xs:anyAtomicType*
```

This function returns a sequence containing all the distinct items that appear in $arg1 but not in $arg2, in an undefined order.

XSLT implementation

```
<xsl:function name="eg:value-except" as="xs:anyAtomicType*">
  <xsl:param name="arg1" as="xs:anyAtomicType*"/>
  <xsl:param name="arg2" as="xs:anyAtomicType*"/>
  <xsl:sequence
     select="fn:distinct-values($arg1[not(.=$arg2)])"/>
</xsl:function>
```

XQuery implementation

```
declare function eg:value-except (
  $arg1 as xs:anyAtomicType*,
  $arg2 as xs:anyAtomicType*) as xs:anyAtomicType*
{
  fn:distinct-values($arg1[not(.=$arg2)])
}
```

D.3 eg:index-of-node

```
eg:index-of-node($seqParam as node()*, $srchParam as node()) as xs:integer*
```

This function returns a sequence of positive integers giving the positions within the sequence `$seqParam` of nodes that are identical to `$srchParam`.

The nodes in the sequence `$seqParam` are compared with `$srchParam` under the rules for the `is` operator. If a node compares identical, then the position of that node in the sequence `$srchParam` is included in the result.

If the value of `$seqParam` is the empty sequence, or if no node in `$seqParam` matches $srchParam, then the empty sequence is returned.

The index is 1-based, not 0-based.

The result sequence is in ascending numeric order.

XSLT implementation

```
<xsl:function name="eg:index-of-node" as="xs:integer*">
  <xsl:param name="sequence" as="node()*"/>
  <xsl:param name="srch" as="node()"/>
  <xsl:for-each select="$sequence">
    <xsl:if test=". is $srch">
       <xsl:sequence select="position()"/>
    </xsl:if>
  </xsl:for-each>
</xsl:function>
```

XQuery implementation

```
declare function eg:index-of-node($sequence as node()*, $srch as node()) as xs:integer*
{
  for $n at $i in $sequence where ($n is $srch) return $i
}
```

D.4 eg:string-pad

eg:string-pad($padString as *xs:string?*, $padCount as *xs:integer*) as *xs:string*

Returns a `xs:string` consisting of a given number of copies of an `xs:string` argument concatenated together.

XSLT implementation

```
<xsl:function name="eg:string-pad" as="xs:string">
  <xsl:param name="padString" as="xs:string?"/>
  <xsl:param name="padCount" as="xs:integer"/>
  <xsl:sequence select="fn:string-join((for $i in 1 to $padCount
                                   return $padString), '')"/>
 </xsl:function>
```

XQuery implementation

```
declare function eg:string-pad (
  $padString as xs:string?,
  $padCount as xs:integer) as xs:string
{
   fn:string-join((for $i in 1 to $padCount return $padString), "")
}
```

This returns the zero-length string if `$padString` is the empty sequence, which is consistent with the general principle that if an `xs:string` argument is the empty sequence it is treated as if it were the zero-length string.

D.5 eg:distinct-nodes-stable

fn:eg:distinct-nodes-stable($arg as *node()**) as *node()**

This function illustrates one possible implementation of a distinct-nodes function. It removes duplicate nodes by identity, preserving the first occurrence of each node.

XPath

```
$arg[empty(subsequence($arg, 1, position()-1) intersect .)]
```

XSLT implementation

```
<xsl:function name="eg:distinct-nodes-stable" as="node()*">
  <xsl:param name="arg" as="node()*"/>
  <xsl:sequence
    select="$arg[empty(subsequence($arg, 1, position()-1) intersect .)]"/> </xsl:function>
```

XQuery implementation

```
declare function distinct-nodes-stable ($arg as node()*) as node()*
{
   for $a at $apos in $arg
   let $before_a := fn:subsequence($arg, 1, $apos - 1)
   where every $ba in $before_a satisfies not($ba is $a)
   return $a
}
```

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

This appendix provides a summary of features defined in this specification whose effect is explicitly ·implementation-defined·. The conformance rules require vendors to provide documentation that explains how these choices have been exercised.

1. The destination of the trace output is ·implementation-defined·. See **4 The Trace Function**.
2. For `xs:integer` operations, implementations that support limited-precision integer operations ·must· either raise an error [err:FOAR0002] or provide an ·implementation-defined· mechanism that allows users to choose between raising an error and returning a result that is modulo the largest representable integer value. See **6.2 Operators on Numeric Values**.
3. For `xs:decimal` values the number of digits of precision returned by the numeric operators is ·implementation-defined·. See **6.2 Operators on Numeric Values**. See also **17.1.3.3 Casting to xs:decimal** and **17.1.3.4 Casting to xs:integer**
4. If the number of digits in the result of a numeric operation exceeds the number of digits that the implementation supports, the result is truncated or rounded in an ·implementation-defined· manner. See **6.2 Operators on Numeric Values**. See also **17.1.3.3 Casting to xs:decimal** and **17.1.3.4 Casting to xs:integer**
5. It is ·implementation-defined· which version of Unicode is supported by the features defined in this specification, but it is recommended that the most recent version of Unicode be used. See **7.1 String Types**.
6. For **7.4.6 fn:normalize-unicode**, conforming implementations ·must· support normalization form "NFC" and ·may· support normalization forms "NFD", "NFKC", "NFKD", "FULLY-NORMALIZED". They ·may· also support other normalization forms with ·implementation-defined· semantics.
7. The ability to decompose strings into collation units suitable for substring matching is an ·implementation-defined· property of a collation. See **7.5 Functions Based on Substring Matching**.
8. All *minimally conforming* processors ·must· support year values with a minimum of 4 digits (i.e., YYYY) and a minimum fractional second precision of 1 millisecond or three digits (i.e., s.sss). However, *conforming processors* ·may· set larger ·implementation-defined· limits on the maximum number of digits they support in these two situations. See **10.1.1 Limits and Precision**.

9. The result of casting a string to `xs:decimal`, when the resulting value is not too large or too small but nevertheless has too many decimal digits to be accurately represented, is implementation-defined. See **17.1.1 Casting from xs:string and xs:untypedAtomic**.
10. Various aspects of the processing provided by **15.5.4 fn:doc** are ·implementation-defined·. Implementations may provide external configuration options that allow any aspect of the processing to be controlled by the user.
11. The manner in which implementations provide options to weaken the ·stable· characteristic of **15.5.6 fn:collection** and **15.5.4 fn:doc** are ·implementation-defined·.

F Changes since the First Edition (Non-Normative)

The changes made to this document are described in detail in the Errata to the first edition. The rationale for each erratum is explained in the corresponding Bugzilla database entry. The following table summarizes the errata that have been applied.

Erratum	Bugzilla	Category	Description
E1	4373	substantive	In fn:resolve-uri it is unclear what happens when the supplied base URI is a relative reference
E2	4384	editorial	The description of fn:subsequence contains a spurious variable $p
E3	4385	markup	An example under fn:idref is incorrectly formatted
E4	4106 4634	substantive	The regex specification allows a back-reference within square brackets, which is meaningless. Furthermore, the specification doesn't say what happens when a regular expression contains a back-reference to a non-existent subexpression.
E5	4448	editorial	The function signatures for the internal functions op:subtract-dates and op:subtract-dateTimes incorrectly allow an empty sequence as the return value.
E6	4471	substantive	Casting from date and time type to string represents the UTC timezone as "+00:00" rather than as "Z". This erratum changes the representation to "Z".
E7	4543	substantive	The meaning of the regex flag "m" is unclear when the last character in the string is a newline
E8	4545	editorial	A character code confuses decimal and hexadecimal notation
E9	4549	editorial	In Appendix D, the function signature of the fn:translate function is quoted incorrectly.
E10	4874	editorial	In 17.1.2, the procedure for casting xs:NOTATION to xs:string does not work because it uses functions that are defined only on xs:QName.
E11	4874	editorial	Although the specification states that a string literal can be cast to an xs:QName or xs:NOTATION, the semantics of the operation are not described in the obvious place. This erratum adds a cross-reference.

Erratum	Bugzilla	Category	Description
E12	4621	substantive	When multiplying or dividing a yearMonthDuration by a number, rounding behavior is underspecified.
E13	4519	editorial	The conditions under which a node has the is-id or is-idref property need to be clarified. (See also corresponding erratum DM.E005 to XDM)
E14	4974	editorial	In fn:normalize-space, a sentence with multiple conditions is ambiguously worded. To solve the problem, the relevant sentence can be simplified, because it doesn't need to say what happens when the argument is "." and there is no context item; that's covered in the rules for evaluating ".".
E15	5235	editorial	In fn:namespace-uri, the terminology "the namespace URI of the xs:QName of $arg" is incorrect. It's not clear that it's referring to the name of the node, rather than (say) its type annotation.
E16	5246	markup	In fn:lang, the list item numbers (1) and (2) are duplicated.
E17	5251	substantive	In fn:starts-with and fn:ends-with, the requirement that there should be a minimal match at the start of the string gives unacceptable results. Any match suffices.
E18	5271	editorial	In the (non-normative) appendix summarizing error conditions, the description of code FORG0008 is misleading.
E19	5284	editorial	Typo in the description of the fn:concat function.
E20	5287	editorial	Errors in examples for the function op:duration-equal.
E21	5597	markup	Errors in examples for the function fn:string-join.
E22	5618	editorial	Narrative for fn:namespace-uri-from-QName refers to xs:string rather than xs:anyURI.
E23	5617	editorial	Summary of op:unary-plus and op:unary-minus ignores the possibility of type promotion.
E24	4106 4634 5348	substantive	The regex specification allows a back-reference within square brackets, which is meaningless. Furthermore, the specification doesn't say what happens when a regular expression contains a back-reference to a non-existent subexpression.
E25	5719	editorial	Misplaced full stop in (non-normative) error text for error FORX0001

Erratum	Bugzilla	Category	Description
E26	5688	substantive	The doc() and doc-available() functions are unclear on the rules for validating the first argument. They also mandate that invalid URIs should always be rejected: this runs against the practice of many implementations, which often allow strings that are not valid URIs to be dereferenced, for example by the use of a catalog. Note: this change indirectly affects the rules for the document() function in XSLT, which refers normatively to the doc() function
E27	5671	editorial	The rules for fn:min() and fn:max() are not entirely clear about the type of the returned result.
E28	5706	editorial	It is unclear what happens when implementation limits are exceeded in casting to xs:gYear or xs:gYearMonth.
E29	6306	substantive	In the description of fn:idref, fn:normalize-space needs to be applied to the string value of the node, not to its typed value.
E30	6212	substantive	The behavior of the idiv operator is unclear in situations involving rounding or overflow.
E31	6028 6591	substantive	The fn:id() function does not have the correct semantics when dealing with ID-valued elements. The resolution of this problem is to retain the behavior of fn:id() as specified, while introducing a new function fn:element-with-id() whose behavior reflects the intended meaning of ID-valued elements. To avoid making existing implementations non-conformant, the new function is optional.
E32	6124	editorial	Code in illustrative functions for casting to dates and times uses fn:length in place of fn:string-length.
E33	6316 6212	editorial	The behaviour of the idiv operator is unclear in situations involving rounding or overflow, and it is not stated clearly what the result of idiv is when the second operand is infinity.
E34	6338	editorial	In fn:string-length, a sentence with multiple conditions is ambiguously worded. To solve the problem, the relevant sentence can be simplified, because it doesn't need to say what happens when the argument is "." and there is no context item; that's covered in the rules for evaluating ".". (See also erratum E14)
E35	6342	editorial	Missing closing quote in example of op:divide-dayTimeDuration-by-dayTimeDuration
E36	6346	editorial	Misleading example of fn:number
E37	6347	editorial	Missing closing parenthesis in description of fn:local-name

Erratum	Bugzilla	Category	Description
E38	6348	editorial	Incorrect duration syntax in example code
E39	6355	editorial	Incorrect example for op:divide-dayTimeDuration (uses wrong type name)
E40	6359	editorial	Incorrect example for op:gMonth-equal (missing closing parenthesis)
E41	6371	editorial	Unclear scenario for example of fn:index-of
E42	6372	substantive	The rules for comparing namespace nodes in fn:deep-equal() are inappropriate, for example they can lead to a node not being equal to itself.
E43	6375	editorial	It is not explicitly stated that notes and examples are non-normative
E44	5183	substantive	The distinct-values() function has problems caused by non-transitivity of the eq operator
E45	6344	editorial	Typographical error in the explanation of an example of op:gYearEqual()
E46	6345	editorial	The word "Summary" is repeated in the specification of op:gMonthDayEqual()
E47	5671	editorial	The rules for fn:min() and fn:max() appear contradictory about whether the input sequence is allowed to contain a mixture of xs:string and xs:anyURI values. (This erratum relates to the problem identified in comment #9 of the Bugzilla entry.)
E48	6591	editorial	The reference to xs:IDREFS in the description of fn:id() is misleading, since xs:IDREFS has a minLength of 1.

G Function and Operator Quick Reference (Non-Normative)

G.1 Functions and Operators by Section

2 Accessors

2.1 fn:node-name

`fn:node-name($arg as node()?) as xs:QName?`

2.2 fn:nilled

`fn:nilled($arg as node()?) as xs:boolean?`

2.3 fn:string

`fn:string() as xs:string`

`fn:string($arg as item()?) as xs:string`

2.4 fn:data

`fn:data($arg as item()*) as xs:anyAtomicType*`

2.5 fn:base-uri

`fn:base-uri() as xs:anyURI?`

`fn:base-uri($arg as node()?) as xs:anyURI?`

2.6 fn:document-uri

fn:document-uri($arg as node()?) as xs:anyURI?

3 The Error Function

fn:error() as none
fn:error($error as xs:QName) as none
fn:error($error as xs:QName?, $description as xs:string) as none
fn:error($error as xs:QName?, $description as xs:string, $error-object as item()*) as none

4 The Trace Function

fn:trace($value as item()*, $label as xs:string) as item()*

5 Constructor Functions

5.2 A Special Constructor Function for xs:dateTime

fn:dateTime($arg1 as xs:date?, $arg2 as xs:time?) as xs:dateTime?

6 Functions and Operators on Numerics

6.2 Operators on Numeric Values

op:numeric-add($arg1 as numeric, $arg2 as numeric) as numeric
op:numeric-subtract($arg1 as numeric, $arg2 as numeric) as numeric
op:numeric-multiply($arg1 as numeric, $arg2 as numeric) as numeric
op:numeric-divide($arg1 as numeric, $arg2 as numeric) as numeric
op:numeric-integer-divide($arg1 as numeric, $arg2 as numeric) as xs:integer
op:numeric-mod($arg1 as numeric, $arg2 as numeric) as numeric
op:numeric-unary-plus($arg as numeric) as numeric
op:numeric-unary-minus($arg as numeric) as numeric

6.3 Comparison Operators on Numeric Values

op:numeric-equal($arg1 as numeric, $arg2 as numeric) as xs:boolean
op:numeric-less-than($arg1 as numeric, $arg2 as numeric) as xs:boolean
op:numeric-greater-than($arg1 as numeric, $arg2 as numeric) as xs:boolean

6.4 Functions on Numeric Values

fn:abs($arg as numeric?) as numeric?
fn:ceiling($arg as numeric?) as numeric?
fn:floor($arg as numeric?) as numeric?
fn:round($arg as numeric?) as numeric?
fn:round-half-to-even($arg as numeric?) as numeric?
fn:round-half-to-even($arg as numeric?, $precision as xs:integer) as numeric?

7 Functions on Strings

7.2 Functions to Assemble and Disassemble Strings

fn:codepoints-to-string($arg as xs:integer*) as xs:string
fn:string-to-codepoints($arg as xs:string?) as xs:integer*

7.3 Equality and Comparison of Strings

fn:compare($comparand1 as xs:string?, $comparand2 as xs:string?) as xs:integer?
fn:compare($comparand1 as xs:string?, $comparand2 as xs:string?,
    $collation as xs:string) as xs:integer?
fn:codepoint-equal($comparand1 as xs:string?, $comparand2 as xs:string?) as xs:boolean?

7.4 Functions on String Values

fn:concat($arg1 as xs:anyAtomicType?, $arg2 as xs:anyAtomicType?, ...) as xs:string
fn:string-join($arg1 as xs:string*, $arg2 as xs:string) as xs:string
fn:substring($sourceString as xs:string?, $startingLoc as xs:double) as xs:string
fn:substring($sourceString as xs:string?, $startingLoc as xs:double,
    $length as xs:double) as xs:string
fn:string-length() as xs:integer
fn:string-length($arg as xs:string?) as xs:integer
fn:normalize-space() as xs:string
fn:normalize-space($arg as xs:string?) as xs:string
fn:normalize-unicode($arg as xs:string?) as xs:string

fn:normalize-unicode($arg as xs:string?, $normalizationForm as xs:string) as xs:string
fn:upper-case($arg as xs:string?) as xs:string
fn:lower-case($arg as xs:string?) as xs:string
fn:translate($arg as xs:string?, $mapString as xs:string,
    $transString as xs:string) as xs:string
fn:encode-for-uri($uri-part as xs:string?) as xs:string
fn:iri-to-uri($iri as xs:string?) as xs:string
fn:escape-html-uri($uri as xs:string?) as xs:string

7.5 Functions Based on Substring Matching

fn:contains($arg1 as xs:string?, $arg2 as xs:string?) as xs:boolean
fn:contains($arg1 as xs:string?, $arg2 as xs:string?,
    $collation as xs:string) as xs:boolean
fn:starts-with($arg1 as xs:string?, $arg2 as xs:string?) as xs:boolean
fn:starts-with($arg1 as xs:string?, $arg2 as xs:string?,
    $collation as xs:string) as xs:boolean
fn:ends-with($arg1 as xs:string?, $arg2 as xs:string?) as xs:boolean
fn:ends-with($arg1 as xs:string?, $arg2 as xs:string?,
    $collation as xs:string) as xs:boolean
fn:substring-before($arg1 as xs:string?, $arg2 as xs:string?) as xs:string
fn:substring-before($arg1 as xs:string?, $arg2 as xs:string?,
    $collation as xs:string) as xs:string
fn:substring-after($arg1 as xs:string?, $arg2 as xs:string?) as xs:string
fn:substring-after($arg1 as xs:string?, $arg2 as xs:string?,
    $collation as xs:string) as xs:string

7.6 String Functions that Use Pattern Matching

fn:matches($input as xs:string?, $pattern as xs:string) as xs:boolean
fn:matches($input as xs:string?, $pattern as xs:string,
    $flags as xs:string) as xs:boolean
fn:replace($input as xs:string?, $pattern as xs:string,
    $replacement as xs:string) as xs:string
fn:replace($input as xs:string?, $pattern as xs:string, $replacement as xs:string,
    $flags as xs:string) as xs:string
fn:tokenize($input as xs:string?, $pattern as xs:string) as xs:string*
fn:tokenize($input as xs:string?, $pattern as xs:string,
    $flags as xs:string) as xs:string*

8 Functions on anyURI

8.1 fn:resolve-uri

fn:resolve-uri($relative as xs:string?) as xs:anyURI?
fn:resolve-uri($relative as xs:string?, $base as xs:string) as xs:anyURI?

9 Functions and Operators on Boolean Values

9.1 Additional Boolean Constructor Functions

fn:true() as xs:boolean
fn:false() as xs:boolean

9.2 Operators on Boolean Values

op:boolean-equal($value1 as xs:boolean, $value2 as xs:boolean) as xs:boolean
op:boolean-less-than($arg1 as xs:boolean, $arg2 as xs:boolean) as xs:boolean
op:boolean-greater-than($arg1 as xs:boolean, $arg2 as xs:boolean) as xs:boolean

9.3 Functions on Boolean Values

fn:not($arg as item()*) as xs:boolean

10 Functions and Operators on Durations, Dates and Times

10.4 Comparison Operators on Duration, Date and Time Values

```
op:yearMonthDuration-less-than($arg1 as xs:yearMonthDuration,
      $arg2 as xs:yearMonthDuration) as xs:boolean
op:yearMonthDuration-greater-than($arg1 as xs:yearMonthDuration,
      $arg2 as xs:yearMonthDuration) as xs:boolean
op:dayTimeDuration-less-than($arg1 as xs:dayTimeDuration,
      $arg2 as xs:dayTimeDuration) as xs:boolean
op:dayTimeDuration-greater-than($arg1 as xs:dayTimeDuration,
      $arg2 as xs:dayTimeDuration) as xs:boolean
op:duration-equal($arg1 as xs:duration, $arg2 as xs:duration) as xs:boolean
op:dateTime-equal($arg1 as xs:dateTime, $arg2 as xs:dateTime) as xs:boolean
op:dateTime-less-than($arg1 as xs:dateTime, $arg2 as xs:dateTime) as xs:boolean
op:dateTime-greater-than($arg1 as xs:dateTime, $arg2 as xs:dateTime) as xs:boolean
op:date-equal($arg1 as xs:date, $arg2 as xs:date) as xs:boolean
op:date-less-than($arg1 as xs:date, $arg2 as xs:date) as xs:boolean
op:date-greater-than($arg1 as xs:date, $arg2 as xs:date) as xs:boolean
op:time-equal($arg1 as xs:time, $arg2 as xs:time) as xs:boolean
op:time-less-than($arg1 as xs:time, $arg2 as xs:time) as xs:boolean
op:time-greater-than($arg1 as xs:time, $arg2 as xs:time) as xs:boolean
op:gYearMonth-equal($arg1 as xs:gYearMonth, $arg2 as xs:gYearMonth) as xs:boolean
op:gYear-equal($arg1 as xs:gYear, $arg2 as xs:gYear) as xs:boolean
op:gMonthDay-equal($arg1 as xs:gMonthDay, $arg2 as xs:gMonthDay) as xs:boolean
op:gMonth-equal($arg1 as xs:gMonth, $arg2 as xs:gMonth) as xs:boolean
op:gDay-equal($arg1 as xs:gDay, $arg2 as xs:gDay) as xs:boolean
```

10.5 Component Extraction Functions on Durations, Dates and Times

```
fn:years-from-duration($arg as xs:duration?) as xs:integer?
fn:months-from-duration($arg as xs:duration?) as xs:integer?
fn:days-from-duration($arg as xs:duration?) as xs:integer?
fn:hours-from-duration($arg as xs:duration?) as xs:integer?
fn:minutes-from-duration($arg as xs:duration?) as xs:integer?
fn:seconds-from-duration($arg as xs:duration?) as xs:decimal?
fn:year-from-dateTime($arg as xs:dateTime?) as xs:integer?
fn:month-from-dateTime($arg as xs:dateTime?) as xs:integer?
fn:day-from-dateTime($arg as xs:dateTime?) as xs:integer?
fn:hours-from-dateTime($arg as xs:dateTime?) as xs:integer?
fn:minutes-from-dateTime($arg as xs:dateTime?) as xs:integer?
fn:seconds-from-dateTime($arg as xs:dateTime?) as xs:decimal?
fn:timezone-from-dateTime($arg as xs:dateTime?) as xs:dayTimeDuration?
fn:year-from-date($arg as xs:date?) as xs:integer?
fn:month-from-date($arg as xs:date?) as xs:integer?
fn:day-from-date($arg as xs:date?) as xs:integer?
fn:timezone-from-date($arg as xs:date?) as xs:dayTimeDuration?
fn:hours-from-time($arg as xs:time?) as xs:integer?
fn:minutes-from-time($arg as xs:time?) as xs:integer?
fn:seconds-from-time($arg as xs:time?) as xs:decimal?
fn:timezone-from-time($arg as xs:time?) as xs:dayTimeDuration?
```

10.6 Arithmetic Operators on Durations

```
op:add-yearMonthDurations($arg1 as xs:yearMonthDuration,
      $arg2 as xs:yearMonthDuration) as xs:yearMonthDuration
op:subtract-yearMonthDurations($arg1 as xs:yearMonthDuration,
      $arg2 as xs:yearMonthDuration) as xs:yearMonthDuration
op:multiply-yearMonthDuration($arg1 as xs:yearMonthDuration,
      $arg2 as xs:double) as xs:yearMonthDuration
op:divide-yearMonthDuration($arg1 as xs:yearMonthDuration,
      $arg2 as xs:double) as xs:yearMonthDuration
op:divide-yearMonthDuration-by-yearMonthDuration($arg1 as xs:yearMonthDuration,
      $arg2 as xs:yearMonthDuration) as xs:decimal
```

op:add-dayTimeDurations($arg1 as xs:dayTimeDuration,
    $arg2 as xs:dayTimeDuration) as xs:dayTimeDuration
op:subtract-dayTimeDurations($arg1 as xs:dayTimeDuration,
    $arg2 as xs:dayTimeDuration) as xs:dayTimeDuration
op:multiply-dayTimeDuration($arg1 as xs:dayTimeDuration,
    $arg2 as xs:double) as xs:dayTimeDuration
op:divide-dayTimeDuration($arg1 as xs:dayTimeDuration,
    $arg2 as xs:double) as xs:dayTimeDuration
op:divide-dayTimeDuration-by-dayTimeDuration($arg1 as xs:dayTimeDuration,
    $arg2 as xs:dayTimeDuration) as xs:decimal

10.7 Timezone Adjustment Functions on Dates and Time Values

fn:adjust-dateTime-to-timezone($arg as xs:dateTime?) as xs:dateTime?
fn:adjust-dateTime-to-timezone($arg as xs:dateTime?,
    $timezone as xs:dayTimeDuration?) as xs:dateTime?
fn:adjust-date-to-timezone($arg as xs:date?) as xs:date?
fn:adjust-date-to-timezone($arg as xs:date?,
    $timezone as xs:dayTimeDuration?) as xs:date?
fn:adjust-time-to-timezone($arg as xs:time?) as xs:time?
fn:adjust-time-to-timezone($arg as xs:time?,
    $timezone as xs:dayTimeDuration?) as xs:time?

10.8 Arithmetic Operators on Durations, Dates and Times

op:subtract-dateTimes($arg1 as xs:dateTime, $arg2 as xs:dateTime) as xs:dayTimeDuration
op:subtract-dates($arg1 as xs:date, $arg2 as xs:date) as xs:dayTimeDuration
op:subtract-times($arg1 as xs:time, $arg2 as xs:time) as xs:dayTimeDuration
op:add-yearMonthDuration-to-dateTime($arg1 as xs:dateTime,
    $arg2 as xs:yearMonthDuration) as xs:dateTime
op:add-dayTimeDuration-to-dateTime($arg1 as xs:dateTime,
    $arg2 as xs:dayTimeDuration) as xs:dateTime
op:subtract-yearMonthDuration-from-dateTime($arg1 as xs:dateTime,
    $arg2 as xs:yearMonthDuration) as xs:dateTime
op:subtract-dayTimeDuration-from-dateTime($arg1 as xs:dateTime,
    $arg2 as xs:dayTimeDuration) as xs:dateTime
op:add-yearMonthDuration-to-date($arg1 as xs:date,
    $arg2 as xs:yearMonthDuration) as xs:date
op:add-dayTimeDuration-to-date($arg1 as xs:date,
    $arg2 as xs:dayTimeDuration) as xs:date
op:subtract-yearMonthDuration-from-date($arg1 as xs:date,
    $arg2 as xs:yearMonthDuration) as xs:date
op:subtract-dayTimeDuration-from-date($arg1 as xs:date,
    $arg2 as xs:dayTimeDuration) as xs:date
op:add-dayTimeDuration-to-time($arg1 as xs:time,
    $arg2 as xs:dayTimeDuration) as xs:time
op:subtract-dayTimeDuration-from-time($arg1 as xs:time,
    $arg2 as xs:dayTimeDuration) as xs:time

11 Functions Related to QNames

11.1 Additional Constructor Functions for QNames

fn:resolve-QName($qname as xs:string?, $element as element()) as xs:QName?
fn:QName($paramURI as xs:string?, $paramQName as xs:string) as xs:QName

11.2 Functions and Operators Related to QNames

op:QName-equal($arg1 as xs:QName, $arg2 as xs:QName) as xs:boolean
fn:prefix-from-QName($arg as xs:QName?) as xs:NCName?
fn:local-name-from-QName($arg as xs:QName?) as xs:NCName?
fn:namespace-uri-from-QName($arg as xs:QName?) as xs:anyURI?
fn:namespace-uri-for-prefix($prefix as xs:string?, $element as element()) as xs:anyURI?
fn:in-scope-prefixes($element as element()) as xs:string*

12 Operators on base64Binary and hexBinary

12.1 Comparisons of base64Binary and hexBinary Values

```
op:hexBinary-equal($value1 as xs:hexBinary, $value2 as xs:hexBinary) as xs:boolean
op:base64Binary-equal($value1 as xs:base64Binary,
      $value2 as xs:base64Binary) as xs:boolean
```

13 Operators on NOTATION

13.1 Operators on NOTATION

```
op:NOTATION-equal($arg1 as xs:NOTATION, $arg2 as xs:NOTATION) as xs:boolean
```

14 Functions and Operators on Nodes

14.1 fn:name

```
fn:name() as xs:string
fn:name($arg as node()?) as xs:string
```

14.2 fn:local-name

```
fn:local-name() as xs:string
fn:local-name($arg as node()?) as xs:string
```

14.3 fn:namespace-uri

```
fn:namespace-uri() as xs:anyURI
fn:namespace-uri($arg as node()?) as xs:anyURI
```

14.4 fn:number

```
fn:number() as xs:double
fn:number($arg as xs:anyAtomicType?) as xs:double
```

14.5 fn:lang

```
fn:lang($testlang as xs:string?) as xs:boolean
fn:lang($testlang as xs:string?, $node as node()) as xs:boolean
```

14.6 op:is-same-node

```
op:is-same-node($parameter1 as node(), $parameter2 as node()) as xs:boolean
```

14.7 op:node-before

```
op:node-before($parameter1 as node(), $parameter2 as node()) as xs:boolean
```

14.8 op:node-after

```
op:node-after($parameter1 as node(), $parameter2 as node()) as xs:boolean
```

14.9 fn:root

```
fn:root() as node()
fn:root($arg as node()?) as node()?
```

15 Functions and Operators on Sequences

15.1 General Functions and Operators on Sequences

```
fn:boolean($arg as item()*) as xs:boolean
op:concatenate($seq1 as item()*, $seq2 as item()*) as item()*
fn:index-of($seqParam as xs:anyAtomicType*,
      $srchParam as xs:anyAtomicType) as xs:integer*
fn:index-of($seqParam as xs:anyAtomicType*, $srchParam as xs:anyAtomicType,
      $collation as xs:string) as xs:integer*
fn:empty($arg as item()*) as xs:boolean
fn:exists($arg as item()*) as xs:boolean
fn:distinct-values($arg as xs:anyAtomicType*) as xs:anyAtomicType*
fn:distinct-values($arg as xs:anyAtomicType*,
      $collation as xs:string) as xs:anyAtomicType*
fn:insert-before($target as item()*, $position as xs:integer,
      $inserts as item()*) as item()*
fn:remove($target as item()*, $position as xs:integer) as item()*
fn:reverse($arg as item()*) as item()*
fn:subsequence($sourceSeq as item()*, $startingLoc as xs:double) as item()*
```

fn:subsequence($sourceSeq as item()*, $startingLoc as xs:double,
$length as xs:double) as item()*
fn:unordered($sourceSeq as item()*) as item()*

15.2 Functions That Test the Cardinality of Sequences

fn:zero-or-one($arg as item()*) as item()?
fn:one-or-more($arg as item()*) as item()+
fn:exactly-one($arg as item()*) as item()

15.3 Equals, Union, Intersection and Except

fn:deep-equal($parameter1 as item()*, $parameter2 as item()*) as xs:boolean
fn:deep-equal($parameter1 as item()*, $parameter2 as item()*,
$collation as string) as xs:boolean
op:union($parameter1 as node()*, $parameter2 as node()*) as node()*
op:intersect($parameter1 as node()*, $parameter2 as node()*) as node()*
op:except($parameter1 as node()*, $parameter2 as node()*) as node()*

15.4 Aggregate Functions

fn:count($arg as item()*) as xs:integer
fn:avg($arg as xs:anyAtomicType*) as xs:anyAtomicType?
fn:max($arg as xs:anyAtomicType*) as xs:anyAtomicType?
fn:max($arg as xs:anyAtomicType*, $collation as string) as xs:anyAtomicType?
fn:min($arg as xs:anyAtomicType*) as xs:anyAtomicType?
fn:min($arg as xs:anyAtomicType*, $collation as string) as xs:anyAtomicType?
fn:sum($arg as xs:anyAtomicType*) as xs:anyAtomicType
fn:sum($arg as xs:anyAtomicType*, $zero as xs:anyAtomicType?) as xs:anyAtomicType?

15.5 Functions and Operators that Generate Sequences

op:to($firstval as xs:integer, $lastval as xs:integer) as xs:integer*
fn:id($arg as xs:string*) as element()*
fn:id($arg as xs:string*, $node as node()) as element()*
fn:idref($arg as xs:string*) as node()*
fn:idref($arg as xs:string*, $node as node()) as node()*
fn:doc($uri as xs:string?) as document-node()?
fn:doc-available($uri as xs:string?) as xs:boolean
fn:collection() as node()*
fn:collection($arg as xs:string?) as node()*
fn:element-with-id($arg as xs:string*) as element()*
fn:element-with-id($arg as xs:string*, $node as node()) as element()*

16 Context Functions

16.1 fn:position

fn:position() as xs:integer

16.2 fn:last

fn:last() as xs:integer

16.3 fn:current-dateTime

fn:current-dateTime() as xs:dateTime

16.4 fn:current-date

fn:current-date() as xs:date

16.5 fn:current-time

fn:current-time() as xs:time

16.6 fn:implicit-timezone

fn:implicit-timezone() as xs:dayTimeDuration

16.7 fn:default-collation

fn:default-collation() as xs:string

16.8 fn:static-base-uri

fn:static-base-uri() as xs:anyURI?

G.2 Functions and Operators Alphabetically

op:numeric-integer-divide($arg1 as numeric, $arg2 as numeric) as xs:integer (§6.2.5)
op:numeric-less-than($arg1 as numeric, $arg2 as numeric) as xs:boolean (§6.3.2)
op:numeric-mod($arg1 as numeric, $arg2 as numeric) as numeric (§6.2.6)
op:numeric-multiply($arg1 as numeric, $arg2 as numeric) as numeric (§6.2.3)
op:numeric-subtract($arg1 as numeric, $arg2 as numeric) as numeric (§6.2.2)
op:numeric-unary-minus($arg as numeric) as numeric (§6.2.8)
op:numeric-unary-plus($arg as numeric) as numeric (§6.2.7)
fn:one-or-more($arg as item()*) as item()+ (§15.2.2)
fn:position() as xs:integer (§16.1)
fn:prefix-from-QName($arg as xs:QName?) as xs:NCName? (§11.2.2)
fn:remove($target as item()*, $position as xs:integer) as item()* (§15.1.8)
fn:replace($input as xs:string?, $pattern as xs:string, $replacement as xs:string) as xs:string (§7.6.3)
fn:replace($input as xs:string?, $pattern as xs:string, $replacement as xs:string, $flags as xs:string) as xs:string (§7.6.3)
fn:resolve-QName($qname as xs:string?, $element as element()) as xs:QName? (§11.1.1)
fn:resolve-uri($relative as xs:string?) as xs:anyURI? (§8.1)
fn:resolve-uri($relative as xs:string?, $base as xs:string) as xs:anyURI? (§8.1)
fn:reverse($arg as item()*) as item()* (§15.1.9)
fn:root() as node() (§14.9)
fn:root($arg as node()?) as node()? (§14.9)
fn:round($arg as numeric?) as numeric? (§6.4.4)
fn:round-half-to-even($arg as numeric?) as numeric? (§6.4.5)
fn:round-half-to-even($arg as numeric?, $precision as xs:integer) as numeric? (§6.4.5)
fn:seconds-from-dateTime($arg as xs:dateTime?) as xs:decimal? (§10.5.12)
fn:seconds-from-duration($arg as xs:duration?) as xs:decimal? (§10.5.6)
fn:seconds-from-time($arg as xs:time?) as xs:decimal? (§10.5.20)
fn:starts-with($arg1 as xs:string?, $arg2 as xs:string?) as xs:boolean (§7.5.2)
fn:starts-with($arg1 as xs:string?, $arg2 as xs:string?, $collation as xs:string) as xs:boolean (§7.5.2)
fn:static-base-uri() as xs:anyURI? (§16.8)
fn:string() as xs:string (§2.3)
fn:string($arg as item()?) as xs:string (§2.3)
fn:string-join($arg1 as xs:string*, $arg2 as xs:string) as xs:string (§7.4.2)
fn:string-length() as xs:integer (§7.4.4)
fn:string-length($arg as xs:string?) as xs:integer (§7.4.4)
fn:string-to-codepoints($arg as xs:string?) as xs:integer* (§7.2.2)
fn:subsequence($sourceSeq as item()*, $startingLoc as xs:double) as item()* (§15.1.10)
fn:subsequence($sourceSeq as item()*, $startingLoc as xs:double, $length as xs:double) as item()* (§15.1.10)
fn:substring($sourceString as xs:string?, $startingLoc as xs:double) as xs:string (§7.4.3)
fn:substring($sourceString as xs:string?, $startingLoc as xs:double, $length as xs:double) as xs:string (§7.4.3)
fn:substring-after($arg1 as xs:string?, $arg2 as xs:string?) as xs:string (§7.5.5)
fn:substring-after($arg1 as xs:string?, $arg2 as xs:string?, $collation as xs:string) as xs:string (§7.5.5)
fn:substring-before($arg1 as xs:string?, $arg2 as xs:string?) as xs:string (§7.5.4)
fn:substring-before($arg1 as xs:string?, $arg2 as xs:string?, $collation as xs:string) as xs:string (§7.5.4)
op:subtract-dateTimes($arg1 as xs:dateTime, $arg2 as xs:dateTime) as xs:dayTimeDuration (§10.8.1)
op:subtract-dates($arg1 as xs:date, $arg2 as xs:date) as xs:dayTimeDuration (§10.8.2)
op:subtract-dayTimeDuration-from-date($arg1 as xs:date, $arg2 as xs:dayTimeDuration) as xs:date (§10.8.11)
op:subtract-dayTimeDuration-from-dateTime($arg1 as xs:dateTime, $arg2 as xs:dayTimeDuration) as xs:dateTime (§10.8.7)
op:subtract-dayTimeDuration-from-time($arg1 as xs:time, $arg2 as xs:dayTimeDuration) as xs:time (§10.8.13)

op:subtract-dayTimeDurations($arg1 as xs:dayTimeDuration,
    $arg2 as xs:dayTimeDuration) as xs:dayTimeDuration (§10.6.7)
op:subtract-times($arg1 as xs:time, $arg2 as xs:time) as xs:dayTimeDuration (§10.8.3)
op:subtract-yearMonthDuration-from-date($arg1 as xs:date,
    $arg2 as xs:yearMonthDuration) as xs:date (§10.8.10)
op:subtract-yearMonthDuration-from-dateTime($arg1 as xs:dateTime,
    $arg2 as xs:yearMonthDuration) as xs:dateTime (§10.8.6)
op:subtract-yearMonthDurations($arg1 as xs:yearMonthDuration,
    $arg2 as xs:yearMonthDuration) as xs:yearMonthDuration (§10.6.2)
fn:sum($arg as xs:anyAtomicType*) as xs:anyAtomicType (§15.4.5)
fn:sum($arg as xs:anyAtomicType*, $zero as xs:anyAtomicType?) as xs:anyAtomicType? (§15.4.5)
op:time-equal($arg1 as xs:time, $arg2 as xs:time) as xs:boolean (§10.4.12)
op:time-greater-than($arg1 as xs:time, $arg2 as xs:time) as xs:boolean (§10.4.14)
op:time-less-than($arg1 as xs:time, $arg2 as xs:time) as xs:boolean (§10.4.13)
fn:timezone-from-date($arg as xs:date?) as xs:dayTimeDuration? (§10.5.17)
fn:timezone-from-dateTime($arg as xs:dateTime?) as xs:dayTimeDuration? (§10.5.13)
fn:timezone-from-time($arg as xs:time?) as xs:dayTimeDuration? (§10.5.21)
op:to($firstval as xs:integer, $lastval as xs:integer) as xs:integer* (§15.5.1)
fn:tokenize($input as xs:string?, $pattern as xs:string) as xs:string* (§7.6.4)
fn:tokenize($input as xs:string?, $pattern as xs:string, $flags as xs:string) as xs:string* (§7.6.4)
fn:trace($value as item()*, $label as xs:string) as item()* (§4)
fn:translate($arg as xs:string?, $mapString as xs:string, $transString as xs:string) as xs:string
    (§7.4.9)
fn:true() as xs:boolean (§9.1.1)
op:union($parameter1 as node()*, $parameter2 as node()*) as node()* (§15.3.2)
fn:unordered($sourceSeq as item()*) as item()* (§15.1.11)
fn:upper-case($arg as xs:string?) as xs:string (§7.4.7)
fn:year-from-date($arg as xs:date?) as xs:integer? (§10.5.14)
fn:year-from-dateTime($arg as xs:dateTime?) as xs:integer? (§10.5.7)
op:yearMonthDuration-greater-than($arg1 as xs:yearMonthDuration,
    $arg2 as xs:yearMonthDuration) as xs:boolean (§10.4.2)
op:yearMonthDuration-less-than($arg1 as xs:yearMonthDuration,
    $arg2 as xs:yearMonthDuration) as xs:boolean (§10.4.1)
fn:years-from-duration($arg as xs:duration?) as xs:integer? (§10.5.1)
fn:zero-or-one($arg as item()*) as item()? (§15.2.1)