

**Tamino**

**XQuery 4 Reference Guide**

Version 10.1

April 2018

This document applies to Tamino Version 10.1 and all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Copyright © 1999-2018 Software AG, Darmstadt, Germany and/or Software AG USA, Inc., Reston, VA, USA, and/or its subsidiaries and/or its affiliates and/or their licensors.

The name Software AG and all Software AG product names are either trademarks or registered trademarks of Software AG and/or Software AG USA, Inc. and/or its subsidiaries and/or its affiliates and/or their licensors. Other company and product names mentioned herein may be trademarks of their respective owners.

Detailed information on trademarks and patents owned by Software AG and/or its subsidiaries is located at <http://softwareag.com/licenses>.

Use of this software is subject to adherence to Software AG's licensing conditions and terms. These terms are part of the product documentation, located at <http://softwareag.com/licenses/> and/or in the root installation directory of the licensed product(s).

This software may include portions of third-party products. For third-party copyright notices, license terms, additional rights or restrictions, please refer to "License Texts, Copyright Notices and Disclaimers of Third-Party Products". For certain specific third-party license restrictions, please refer to section E of the Legal Notices available under "License Terms and Conditions for Use of Software AG Products / Copyright and Trademark Notices of Software AG Products". These documents are part of the product documentation, located at <http://softwareag.com/licenses> and/or in the root installation directory of the licensed product(s).

Use, reproduction, transfer, publication or disclosure is prohibited except as specifically provided for in your License Agreement with Software AG.

**Document ID: INS-XQUERY-REF-101-20180413**

## Table of Contents

Preface .....	ix
I .....	1
1 Introduction .....	3
2 Lexical Structure .....	5
Characters .....	6
Identifiers .....	6
Literals .....	6
Comments .....	8
Other Defined Tokens .....	8
II Syntax Constructs .....	9
3 Syntax Constructs Ordered by Categories .....	11
Basic and Primary Expressions .....	12
Path Expressions and Sequence Types .....	12
Sequence Expressions .....	13
Arithmetic, Comparison and Logical Expressions .....	14
Constructors .....	14
Conditional Expressions .....	16
Order-Related Expressions .....	16
Modules, Prologs and Pragmas .....	16
Update Expressions .....	17
Version Declaration .....	18
4 Syntax Constructs in Alphabetical Order .....	19
AbbreviatedStep .....	26
AbsolutePathExpr .....	28
AdditiveExpr .....	29
AndExpr .....	30
AnyKindTest .....	31
AttributeList .....	32
AttributeTest .....	33
AttributeValue .....	34
AttributeValueContent .....	35
Axis .....	37
AxisStep .....	39
CommentTest .....	40
CompCommentConstructor .....	41
CompExpr .....	42
CompPIConstructor .....	43
CompTextConstructor .....	44
ComputedAttributeConstructor .....	45
ComputedDocumentConstructor .....	47
ComputedElementConstructor .....	49
Constructor .....	51
DefaultCollationDecl .....	52

DefaultNamespaceDecl .....	54
DeleteClause .....	56
DirectCommentConstructor .....	57
ElementConstructor .....	58
ElementContent .....	60
ElementNameOrFunctionCall .....	62
ElementTest .....	63
EnclosedExpr .....	64
Expr .....	65
ExprSequence .....	67
FLWORExpr .....	68
FLWUExpr .....	71
ForClause .....	73
FunctionDecl .....	74
GeneralComp .....	76
GeneralStep .....	78
IfExpr .....	79
InsertClause .....	81
IntersectExceptExpr .....	83
ItemType .....	84
KindTest .....	85
LetClause .....	86
LibraryModule .....	87
Literal .....	88
MainModule .....	89
ModuleDecl .....	90
ModuleImport .....	91
MultiplicativeExpr .....	93
NamespaceDecl .....	95
NameTest .....	97
NoAxisStep .....	98
NodeComp .....	99
NodeTest .....	101
NumericLiteral .....	102
OrderByClause .....	103
OrExpr .....	105
ParenthesizedExpr .....	106
PathExpr .....	107
Pragma .....	108
PrimaryExpr .....	109
ProcessingInstructionTest .....	111
Prolog .....	112
RangeExpr .....	114
RelativePathExpr .....	115
RenameClause .....	117

ReplaceClause .....	118
SequenceType .....	119
SerializationSpec .....	120
SortExpr .....	125
SortSpecList .....	127
StepExpr .....	129
StepQualifiers .....	130
TaminoQPIExecution .....	132
TaminoQPIExplain .....	134
TaminoQPIInline .....	136
TaminoQPIOptimization .....	137
TextTest .....	138
TreatExpr .....	139
UnaryExpr .....	140
UnionExpr .....	142
UpdateExpr .....	143
UpdateIfExpr .....	144
UpdateSequence .....	146
UpdatingFunction .....	147
UpdatingFunctionDecl .....	148
ValueComp .....	149
VarDecl .....	151
VersionDecl .....	153
WhereClause .....	154
Wildcard .....	155
XmlProcessingInstruction .....	157
XQueryModule .....	158
III Functions .....	159
5 Functions Ordered by Categories .....	161
Tamino Functions .....	162
Tamino WebDAV Functions .....	162
Text Retrieval Functions .....	163
XQuery Functions .....	163
6 Functions in Alphabetical Order .....	171
fn:abs .....	183
fn:avg .....	184
fn:boolean .....	185
fn:ceiling .....	186
fn:collection .....	188
fn:compare .....	189
fn:concat .....	191
fn:contains .....	192
fn:count .....	194
fn:current-date .....	195
fn:current-dateTime .....	196

fn:current-time .....	197
fn:data .....	198
fn:day-from-date .....	199
fn:day-from-dateTime .....	200
fn:deep-equal .....	201
fn:distinct-values .....	204
fn:ends-with .....	206
fn:expanded-QName .....	208
fn:false .....	209
fn:floor .....	210
fn:get-local-name-from-QName .....	212
fn:get-namespace-from-QName .....	213
fn:hours-from-dateTime .....	214
fn:hours-from-time .....	215
fn:id .....	216
fn:idref .....	217
fn:last .....	218
fn:local-name .....	219
fn:lower-case .....	220
fn:matches .....	221
fn:max .....	223
fn:min .....	224
fn:minutes-from-dateTime .....	225
fn:minutes-from-time .....	226
fn:month-from-date .....	227
fn:month-from-dateTime .....	228
fn:namespace-uri .....	229
fn:node-name .....	230
fn:normalize-space .....	231
fn:not .....	232
fn:position .....	233
fn:put .....	234
fn:replace .....	236
fn:reverse .....	238
fn:root .....	239
fn:round .....	240
fn:seconds-from-dateTime .....	242
fn:seconds-from-time .....	243
fn:starts-with .....	244
fn:string .....	245
fn:string-join .....	247
fn:string-length .....	249
fn:subsequence .....	250
fn:substring .....	252
fn:substring-after .....	254

fn:substring-before .....	256
fn:sum .....	258
fn:tokenize .....	259
fn:true .....	260
fn:upper-case .....	261
fn:year-from-date .....	262
fn:year-from-dateTime .....	263
ft:proximity-contains .....	264
ft:text-contains .....	267
tdf:getProperties .....	269
tdf:getProperty .....	270
tdf:isDescendantOf .....	271
tdf:mkcol .....	273
tdf:resource .....	274
tdf:setProperty .....	275
tf:broaderTerm .....	276
tf:broaderTerms .....	278
tf:containsAdjacentText .....	280
tf:containsNearText .....	282
tf:containsText .....	284
tf:content-type .....	286
tf:createAdjacentTextReference .....	287
tf:createNearTextReference .....	289
tf:createNodeReference .....	291
tf:createTextNode .....	292
tf:createTextReference .....	294
tf:document .....	297
tf:getCollation .....	298
tf:getCollection .....	300
tf:get-current-user .....	301
tf:getDocname .....	302
tf:getInoId .....	303
tf:getLastModified .....	304
tf:highlight .....	306
tf:narrowerTerm .....	308
tf:narrowerTerms .....	310
tf:nonXML-kind .....	312
tf:parse .....	313
tf:phonetic .....	314
tf:query .....	316
tf:serialize .....	317
tf:setDocname .....	318
tf:stem .....	319
tf:synonym .....	321
tf:text-content .....	323

xs:anyURI .....	324
xs:base64Binary .....	325
xs:boolean .....	326
xs:byte .....	328
xs:date .....	330
xs:dateTime .....	332
xs:decimal .....	334
xs:double .....	335
xs:duration .....	336
xs:ENTITY .....	338
xs:float .....	340
xs:gDay .....	341
xs:gMonth .....	342
xs:gMonthDay .....	343
xs:gYear .....	344
xs:gYearMonth .....	346
xs:hexBinary .....	348
xs:ID .....	349
xs:IDREF .....	351
xs:int .....	353
xs:integer .....	354
xs:language .....	355
xs:long .....	356
xs:Name .....	358
xs:NCName .....	360
xs:NMTOKEN .....	362
xs:negativeInteger .....	364
xs:nonNegativeInteger .....	366
xs:nonPositiveInteger .....	368
xs:normalizedString .....	370
xs:positiveInteger .....	371
xs:short .....	373
xs:string .....	375
xs:time .....	376
xs:token .....	377
xs:unsignedByte .....	378
xs:unsignedInt .....	380
xs:unsignedLong .....	382
xs:unsignedShort .....	384
Index .....	387



---

# Preface

---

This document contains the reference documentation of Tamino XQuery 4.

You should be familiar with the concepts presented in the XQuery 4 User Guide as well as with the built-in data types defined in [XML Schema](#).

This documentation covers the following topics:

## Introduction

## Lexical Structure

- Syntax Constructs
  - Syntax Constructs Ordered by Categories
  - Syntax Constructs in Alphabetical Order

- Functions
  - Functions Ordered by Categories
  - Functions in Alphabetical Order

---

# I

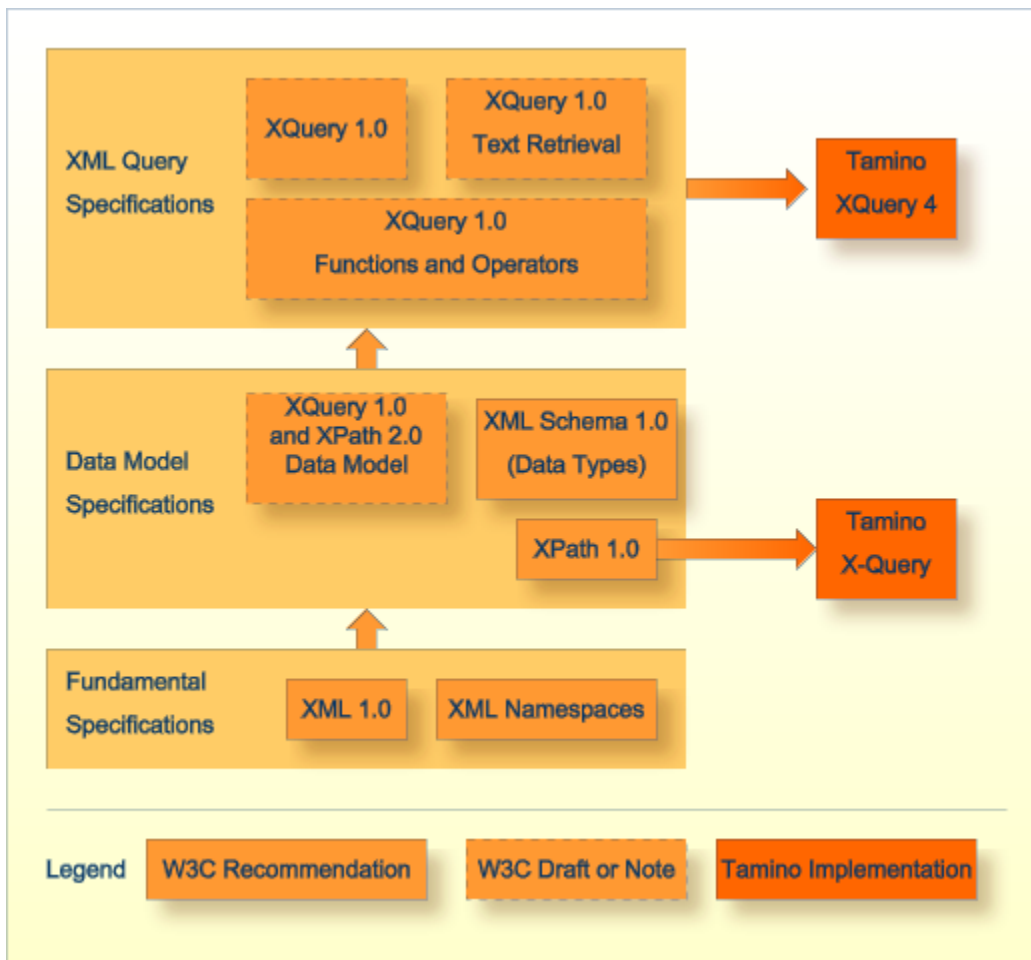
---

■ 1 Introduction .....	3
■ 2 Lexical Structure .....	5



# 1 Introduction

---



## Tamino XQuery4 and W3C Specifications in Context

From the graphics, you can see that Tamino XQuery was built upon a number of XML specifications defined by the World Wide Web Consortium (W3C). Those specifications that treat data model

aspects of XQuery and define the language itself were still in draft status during the time Tamino XQuery was crafted. This is, the specifications had not been promoted as normative recommendations by then. The Tamino implementation of XQuery, called Tamino XQuery 4, followed the draft specifications as closely as possible. At first, it implemented a subset that was large enough to meet general query requirements, but also extended it by including functionality that had already been present in the implementation of its XPath 1.0-based predecessor X-Query.

This documentation refers to the suite of XQuery 1.0 specifications, published as working drafts on August 16, 2002. Any links to these specifications lead to that version. If a link refers to a draft published at another date, there is an explicit hint.



**Note:** In this documentation, the unqualified term “XQuery” is used for the Tamino implementation, which is officially called “Tamino XQuery 4”. The W3C draft specification *as of August 16, 2002* is referred to as “W3C XQuery”.

## 2 Lexical Structure

---

▪ Characters .....	6
▪ Identifiers .....	6
▪ Literals .....	6
▪ Comments .....	8
▪ Other Defined Tokens .....	8

This chapter gives an overview of the syntactic structures on the lexical level of Tamino XQuery 4. Any grammar productions use the same form of BNF that is also used in the respective W3C specifications.

## Characters

---

At the character level, Tamino supports the basic multilingual plane (BMP) and plane 16 as defined in Unicode 3.1. You can find detailed information about Unicode support in the section Unicode and Text Retrieval. In Tamino, a character is defined as follows:

```
Char      ::= #x9 | #xA | #xD | [#x0020-#xD7FF] | [#xE000-#xFFFD] | ↵  
[#x10000-#10FFFF]
```

```
WhitespaceChar ::= #x9 | #xA | #xD | #x20
```

## Identifiers

---

### NCName

An NCName is an XML name with the exception that it may not contain the colon character. It is defined in the W3C recommendation [Namespaces in XML](#), section 2. An XML name is defined as Name in the W3C recommendation [Extensible Markup Language \(XML\) 1.0](#), section 2.4.

### QName

A QName is a name that can be qualified with an optional namespace prefix. It is defined in the W3C recommendation [Namespaces in XML](#), section 3.

## Literals

---

A literal is the direct syntactic representation of an atomic value. In XQuery, the following literals for strings and numbers are defined:



## String Literals

```
StringLiteral ::= "'" ( [^"] | "'" )* "'" | '"' ( [^'] | '"' )* '"'
```

A string literal is a singleton sequence composed of single characters. It has an atomic value of type `xs:string` that is delimited by single or double quotation marks. If you want to use a single quotation mark in a string literal, either delimit the string literal with double quotation marks, or use two consecutive single quotation marks to represent the single quotation mark. The same applies, *mutatis mutandis*, to the double quotation mark.

```
"This is Joe's house."
'This is Joe''s house.'
'I bought a 21" monitor today.'
"He's 5'11" tall."
"He's 5'11"" tall."
```

## Numeric Literals

```
IntegerLiteral ::= [0-9]+
```

```
DecimalLiteral ::= ("." [0-9]+) | ([0-9]+ "." [0-9]*)
```

```
DoubleLiteral ::= (( "." [0-9]+ ) | ([0-9]+ ( "." [0-9]*)? )) ( "e" | "E" ) ( "+" | "-" )? [0-9]+
```

A numeric literal is a singleton sequence with an atomic value whose type is determined as follows:

- If the value contains no "." and no "e" or "E" character, it is of type `xs:integer`.
- If the value contains ".", but no "e" or "E" character, it is of type `xs:decimal`.
- If the value contains an "e" or "E" character, it is of type `xs:double`.

These are valid numeric literals: 42 0.07 .07 3.141 12E04

These are invalid numeric literals: "42" 3,141 0E0A

## Boolean Values

You can represent the Boolean values `true` and `false` by using the built-in functions `fn:true()` and `fn:false()`.

## Using Constructor Functions

You can use constructor functions for the supported types. The name of the constructor function is normally the same as the name of the type.

## Comments

---

A comment in XQuery is defined as follows:

```
ExprComment ::= "{--" ([^}])* "--}"
```

So an XQuery comment is delimited by `{--` and `--}` and it must not contain `}` in between. Comments may even be used within expressions and before or after tokens. However, you can not nest comments. The following query is valid:

```
declare namespace tf="http://namespaces.softwareag.com/tamino/TaminoFunction"
{--
    This query uses a Tamino-specific function.
    Multi-line comments are perfectly valid.
--}
for $a in input{-- you should not put any comments here,
but it still works --}()
return tf:getLastModified({-- no error even here --} $a)
```

You should not place a comment immediately after the opening brace of an enclosed expression, since the character sequence `{{` will be interpreted as an escaped `{`.

## Other Defined Tokens

---

The following tokens are also defined in the XQuery grammar.

```
PredefinedEntityRef ::= "&" ("lt" | "gt" | "amp" | "quot" | "apos") ";"
```

```
Variable ::= "$" QName
```

```
CharRef ::= "&#" ([0-9]+ | ("x" ([0-9] | [a-f] | [A-F]))+) ";"
```

Please note that you can use any of the XQuery keywords as the name of an XML element or attribute, if you prepend a colon:

```
for $a in <for><where let="d">abc</where></for>
return $a/:where[@:let]
```

## II Syntax Constructs

---

This chapter describes the syntax constructs available in Tamino XQuery 4. Wherever possible, the names correspond to the language constructs of W3C XQuery. A syntax diagram illustrates the definition as it appears in the respective production rule of the XQuery grammar. The description is followed by one or more examples.

The links below lead to tables listing the available syntax constructs, along with the corresponding construct in the W3C draft. The link in the W3C column leads you to the production rule in the W3C specification draft of August 16, 2002. Remember that the *current* state of the W3C specification draft may look different. You can reference the constructs in two ways:

- [Syntax Constructs Ordered By Categories](#)
- [Syntax Constructs in Alphabetical Order](#)

### Notation and Usage

---

Syntax diagrams are used to illustrate the grammar rules for the constructs.

- Green boxes denote terminal symbols or string literals.
- Blue boxes denote non-terminal symbols and are linked to the corresponding reference page. You can reach a reference page by choosing the box.

In the example sections you will find parts of sample query expressions rendered bold. They indicate that part of the query which is the actual subject of discussion. However, if this is the complete query, it is not rendered in bold. All query expressions refer to either the patient database or to the XMP database that is taken from the W3C draft [XML Query Use Cases](#). See the Examples section for details about the sample data and any schema files.



# 3

## Syntax Constructs Ordered by Categories

---

■ Basic and Primary Expressions .....	12
■ Path Expressions and Sequence Types .....	12
■ Sequence Expressions .....	13
■ Arithmetic, Comparison and Logical Expressions .....	14
■ Constructors .....	14
■ Conditional Expressions .....	16
■ Order-Related Expressions .....	16
■ Modules, Prologs and Pragmas .....	16
■ Update Expressions .....	17
■ Version Declaration .....	18

## Basic and Primary Expressions

Type	Tamino XQuery	W3C XQuery Draft	Short Description
Basic and Primary Expression	<a href="#">Expr</a>	<a href="#">Expr</a> , <a href="#">TreatExpr</a>	basic XQuery expressions
	<a href="#">PrimaryExpr</a>	<a href="#">PrimaryExpr</a>	basic XQuery primitive
Literals	<a href="#">Literal</a>	<a href="#">Literal</a>	numeric or string literal
	<a href="#">NumericLiteral</a>	<a href="#">NumericLiteral</a>	numeric literal
Parenthesized Expressions	<a href="#">ParenthesizedExpr</a>	<a href="#">ParenthesizedExpr</a>	expression sequence enclosed in parentheses
Function Calls	<a href="#">ElementNameOrFunctionCall</a>	<a href="#">FunctionCall</a>	represent an element or call a function

## Path Expressions and Sequence Types

Type		Tamino XQuery	W3C XQuery Draft	Short Description
Path Expressions		PathExpr	PathExpr	select nodes on a path
		AbsolutePathExpr	subsumed in PathExpr	select nodes on the absolute path
		RelativePathExpr *	RelativePathExpr	select nodes on the relative path
Steps		StepExpr	StepExpr	create and possibly filter a sequence of items
	with axes	AxisStep	ForwardStep, ReverseStep	create and possibly filter a sequence of nodes
		Axis	ForwardAxis, ReverseAxis	direction of movement on a path
		AbbreviatedStep	AbbreviatedForwardStep, AbbreviatedReverseStep	select context, parent, or attribute node using abbreviated syntax
	without axes	NoAxisStep	—	create and possibly filter node sequence on current axis
	others	GeneralStep	subsumed in StepExpr	create and possibly filter item sequence

Type	Tamino XQuery	W3C XQuery Draft	Short Description
	<a href="#">StepQualifiers</a>	<a href="#">Predicates</a>	restrict item sequence to those matching one or more predicates
Sequence Type Syntax	<a href="#">SequenceType</a>	<a href="#">SequenceType</a> ( <i>Draft of October 29, 2004</i> )	specify the type of an XQuery value
	<a href="#">ItemType</a>	<a href="#">ItemType</a> ( <i>Draft of October 29, 2004</i> )	specify the type of an item
Expressions on Sequence Types	<a href="#">TreatExpr</a>	<a href="#">TreatExpr</a> ( <i>Draft of October 29, 2004</i> )	modify static type of operand
Node Tests	<a href="#">NodeTest</a>	<a href="#">NodeTest</a>	check if a node satisfies conditions on the kind or name
	<a href="#">NameTest</a>	<a href="#">NameTest</a>	select nodes by name or wildcard
	<a href="#">Wildcard</a>	<a href="#">Wildcard</a>	select nodes matching the principal node kind
	<a href="#">KindTest</a>	<a href="#">KindTest</a>	check the kind of node
	<a href="#">ProcessingInstructionTest</a>	<a href="#">ProcessingInstructionTest</a>	check for processing instruction nodes
	<a href="#">CommentTest</a>	<a href="#">CommentTest</a>	check for comment nodes
	<a href="#">TextTest</a>	<a href="#">TextTest</a>	check for text nodes
	<a href="#">AnyKindTest</a>	<a href="#">AnyKindTest</a>	check for any kind of node
	<a href="#">AttributeTest</a>	<a href="#">AttributeTest</a> ( <i>Draft of October 29, 2004</i> )	check for attribute nodes
	<a href="#">ElementTest</a>	<a href="#">ElementTest</a> ( <i>Draft of October 29, 2004</i> )	check for element nodes

## Sequence Expressions

Type	Tamino XQuery	W3C XQuery Draft	Short Description
Constructing Sequences	<a href="#">ExprSequence</a>	<a href="#">ExprSequence</a>	construct a sequence of expressions
	<a href="#">ParenthesizedExpr</a>	<a href="#">ParenthesizedExpr</a>	expression sequence enclosed in parentheses
	<a href="#">RangeExpr</a>	<a href="#">RangeExpr</a>	construct a sequence of consecutive integers

Type	Tamino XQuery	W3C XQuery Draft	Short Description
Combining Sequences	<a href="#">UnionExpr</a>	<a href="#">UnionExpr</a>	combine node sequences
	<a href="#">IntersectExceptExpr</a>	<a href="#">IntersectExceptExpr</a>	combine node sequences

## Arithmetic, Comparison and Logical Expressions

Type	Tamino XQuery	W3C XQuery Draft	Short Description
Arithmetic Expressions	<a href="#">AdditiveExpr</a>	<a href="#">AdditiveExpr</a>	add or subtract numerical values
	<a href="#">MultiplicativeExpr</a>	<a href="#">MultiplicativeExpr</a>	multiply or divide numerical values
	<a href="#">UnaryExpr</a>	<a href="#">UnaryExpr</a>	negate numerical value of an expression
Comparison Expressions	<a href="#">CompExpr</a>	<a href="#">ComparisonExpr</a>	general comparison expression
	<a href="#">GeneralComp</a>	<a href="#">GeneralComp</a>	compare sequences
	<a href="#">ValueComp</a>	<a href="#">ValueComp</a>	compare single values
	<a href="#">NodeComp</a>	<a href="#">NodeComp</a>	compare two nodes
Logical Expressions	<a href="#">OrExpr</a>	<a href="#">OrExpr</a>	check if one or both of two expressions are true
	<a href="#">AndExpr</a>	<a href="#">AndExpr</a>	check if both of two expressions are true

## Constructors

Type	Tamino XQuery	W3C XQuery Draft	Short Description
Generic	<a href="#">Constructor</a>	<a href="#">Constructor</a>	constructor expression
Direct Constructors	<a href="#">ElementConstructor</a>	<a href="#">ElementConstructor</a>	construct an XML element
	<a href="#">ElementContent</a>	<a href="#">ElementContent</a>	define content of an element in construction
	<a href="#">AttributeList</a>	<a href="#">AttributeList</a>	create a list of attributes
	<a href="#">AttributeValue</a>	<a href="#">AttributeValue</a>	define value of an attribute in construction
	<a href="#">AttributeValueContent</a>	<a href="#">AttributeValueContent</a>	define value content of an attribute in construction



Type	Tamino XQuery	W3C XQuery Draft	Short Description
	<a href="#">EnclosedExpr</a>	<a href="#">EnclosedExpr</a>	evaluate a sequence of expressions
	<a href="#">DirectCommentConstructor</a>	<a href="#">DirCommentConstructor</a> ( <i>Draft of October 29, 2004</i> )	construct an XML comment
	<a href="#">XmlProcessingInstruction</a>	<a href="#">DirPIConstructor</a> ( <i>Draft of October 29, 2004</i> )	construct an XML processing instruction
Computed Constructors	<a href="#">ComputedElementConstructor</a>	<a href="#">ComputedElementConstructor</a>	construct an element by computing an expression
	<a href="#">ComputedAttributeConstructor</a>	<a href="#">ComputedAttributeConstructor</a>	construct an attribute by computing an expression
	<a href="#">ComputedDocumentConstructor</a>	<a href="#">CompDocConstructor</a>	construct a document by computing an expression
	<a href="#">CompTextConstructor</a>	<a href="#">CompTextConstructor</a> ( <i>Draft of October 29, 2004</i> )	construct a text node by computing an expression
	<a href="#">CompCommentConstructor</a>	<a href="#">CompCommentConstructor</a> ( <i>Draft of October 29, 2004</i> )	construct an XML comment by computing an expression
	<a href="#">CompPIConstructor</a>	<a href="#">CompPIConstructor</a> ( <i>Draft of October 29, 2004</i> )	construct a processing instruction by computing an expression
FLWOR Expressions	<a href="#">FLWORExpr</a>	<a href="#">FLWORExpr</a>	iterate over a sequence of items
	<a href="#">ForClause</a>	<a href="#">ForClause</a>	bind variables by evaluating expression
	<a href="#">LetClause</a>	<a href="#">LetClause</a>	bind variables by evaluating expression
	<a href="#">WhereClause</a>	<a href="#">WhereClause</a>	filter intermediate results

Type	Tamino XQuery	W3C XQuery Draft	Short Description
	<a href="#">OrderByClause</a>	<a href="#">OrderByClause</a> ( <i>Draft of October 29, 2004</i> )	order intermediate results

## Conditional Expressions

Type	Tamino XQuery	W3C XQuery Draft	Short Description
Conditional Expression	<a href="#">IfExpr</a>	<a href="#">IfExpr</a> ( <i>Draft of October 29, 2004</i> )	conditional expression based on <code>if</code> , <code>then</code> , and <code>else</code>

## Order-Related Expressions

Type	Tamino XQuery	W3C XQuery Draft	Short Description
Sort Expressions	<a href="#">SortExpr</a>	<a href="#">SortExpr</a>	sort sequence of items
	<a href="#">SortSpecList</a>	<a href="#">SortSpecList</a> , <a href="#">SortModifier</a>	define ordering expression and sort direction

## Modules, Prologs and Pragmas

Most of the following syntax constructs implement concepts added only recently to the XQuery specification draft. If not otherwise noted they follow the specification of October 29, 2004.

Type	Tamino XQuery	W3C XQuery Draft	Short Description
Modules	<a href="#">XQueryModule</a>	<a href="#">Module</a>	top-level XQuery syntax construct
	<a href="#">MainModule</a>	<a href="#">MainModule</a>	obligatory XQuery module with query body
	<a href="#">LibraryModule</a>	<a href="#">LibraryModule</a>	provide XQuery fragment to other modules
	<a href="#">ModuleDecl</a>	<a href="#">ModuleDecl</a>	identify library module
Prologs	<a href="#">Prolog</a>	<a href="#">Prolog</a>	create environment for query processing
	<a href="#">DefaultCollationDecl</a>	<a href="#">DefaultCollationDecl</a> ( <i>Draft of November 12, 2003</i> )	declare a default collation
	<a href="#">ModuleImport</a>	<a href="#">ModuleImport</a>	import one or more library modules

Type	Tamino XQuery	W3C XQuery Draft	Short Description
	<a href="#">NamespaceDecl</a>	<a href="#">NamespaceDecl</a> ( <i>Draft of November 12, 2003</i> )	declare a namespace
	<a href="#">DefaultNamespaceDecl</a>	<a href="#">DefaultNamespaceDecl</a> ( <i>Draft of November 12, 2003</i> )	declare a default (function) namespace
	<a href="#">VarDecl</a>	<a href="#">VarDecl</a>	declare a variable in query prolog
	<a href="#">FunctionDecl</a>	<a href="#">FunctionDecl</a>	declare a user-defined function
	<a href="#">UpdatingFunctionDecl</a>	—	declaration of an updating function
Pragmas	<a href="#">Pragma</a>	<a href="#">Pragma</a>	query processor directive
	<a href="#">SerializationSpec</a>	<a href="#">XSLT 2.0 and XQuery 1.0 Serialization Specification</a> ( <i>Draft of November 12, 2003</i> )	specify serialization of query output
	<a href="#">TaminoQPIExecution</a>	—	determine parameters of query execution
	<a href="#">TaminoQPIExplain</a>	—	retrieve query execution plan
	<a href="#">TaminoQPIInline</a>	—	use default inlining strategy for user-defined functions
	<a href="#">TaminoQPIOptimization</a>	—	choose query optimization strategy

## Update Expressions

Type	Tamino XQuery	W3C XQuery Draft	Short Description
FLWU Expression	<a href="#">FLWUExpr</a>	—	perform one or more update operations
Updating Sequences	<a href="#">UpdateSequence</a>	—	sequence of basic update expressions
	<a href="#">UpdateExpr</a>	—	update expression
	<a href="#">UpdateIfExpr</a>	—	conditional update expression
	<a href="#">UpdatingFunction</a>	—	updating function invocation
	<a href="#">InsertClause</a>	—	insert a node sequence at update position
	<a href="#">DeleteClause</a>	—	delete node sequence
	<a href="#">RenameClause</a>	—	rename the nodes of a node sequence
	<a href="#">ReplaceClause</a>	—	replace a node

## Version Declaration

---

Tamino XQuery	W3C XQuery Draft	Short Description
<code>VersionDecl</code>	—	specifies the version of the XQuery language — reserved for future use

# 4

## Syntax Constructs in Alphabetical Order

---

▪ AbbreviatedStep .....	26
▪ AbsolutePathExpr .....	28
▪ AdditiveExpr .....	29
▪ AndExpr .....	30
▪ AnyKindTest .....	31
▪ AttributeList .....	32
▪ AttributeTest .....	33
▪ AttributeValue .....	34
▪ AttributeValueContent .....	35
▪ Axis .....	37
▪ AxisStep .....	39
▪ CommentTest .....	40
▪ CompCommentConstructor .....	41
▪ CompExpr .....	42
▪ CompPIConstructor .....	43
▪ CompTextConstructor .....	44
▪ ComputedAttributeConstructor .....	45
▪ ComputedDocumentConstructor .....	47
▪ ComputedElementConstructor .....	49
▪ Constructor .....	51
▪ DefaultCollationDecl .....	52
▪ DefaultNamespaceDecl .....	54
▪ DeleteClause .....	56
▪ DirectCommentConstructor .....	57
▪ ElementConstructor .....	58
▪ ElementContent .....	60
▪ ElementNameOrFunctionCall .....	62
▪ ElementTest .....	63
▪ EnclosedExpr .....	64
▪ Expr .....	65
▪ ExprSequence .....	67
▪ FLWORExpr .....	68

▪ FLWUExpr .....	71
▪ ForClause .....	73
▪ FunctionDecl .....	74
▪ GeneralComp .....	76
▪ GeneralStep .....	78
▪ IfExpr .....	79
▪ InsertClause .....	81
▪ IntersectExceptExpr .....	83
▪ ItemType .....	84
▪ KindTest .....	85
▪ LetClause .....	86
▪ LibraryModule .....	87
▪ Literal .....	88
▪ MainModule .....	89
▪ ModuleDecl .....	90
▪ ModuleImport .....	91
▪ MultiplicativeExpr .....	93
▪ NamespaceDecl .....	95
▪ NameTest .....	97
▪ NoAxisStep .....	98
▪ NodeComp .....	99
▪ NodeTest .....	101
▪ NumericLiteral .....	102
▪ OrderByClause .....	103
▪ OrExpr .....	105
▪ ParenthesizedExpr .....	106
▪ PathExpr .....	107
▪ Pragma .....	108
▪ PrimaryExpr .....	109
▪ ProcessingInstructionTest .....	111
▪ Prolog .....	112
▪ RangeExpr .....	114
▪ RelativePathExpr .....	115
▪ RenameClause .....	117
▪ ReplaceClause .....	118
▪ SequenceType .....	119
▪ SerializationSpec .....	120
▪ SortExpr .....	125
▪ SortSpecList .....	127
▪ StepExpr .....	129
▪ StepQualifiers .....	130
▪ TaminoQPIExecution .....	132
▪ TaminoQPIExplain .....	134
▪ TaminoQPIInline .....	136
▪ TaminoQPIOptimization .....	137

▪ TextTest .....	138
▪ TreatExpr .....	139
▪ UnaryExpr .....	140
▪ UnionExpr .....	142
▪ UpdateExpr .....	143
▪ UpdateIfExpr .....	144
▪ UpdateSequence .....	146
▪ UpdatingFunction .....	147
▪ UpdatingFunctionDecl .....	148
▪ ValueComp .....	149
▪ VarDecl .....	151
▪ VersionDecl .....	153
▪ WhereClause .....	154
▪ Wildcard .....	155
▪ XmlProcessingInstruction .....	157
▪ XQueryModule .....	158

Tamino XQuery Syntax Construct	W3C XQuery Expression	Short Description
AbbreviatedStep	AbbreviatedForwardStep, AbbreviatedReverseStep	select context, parent, or attribute node using abbreviated syntax
AbsolutePathExpr	subsumed in PathExpr	select nodes on the absolute path
AdditiveExpr	AdditiveExpr	add or subtract numerical values
AndExpr	AndExpr	check if both of two expressions are logically true
AnyKindTest	AnyKindTest	check for any kind of node
AttributeList	AttributeList	create a list of attributes
AttributeTest	AttributeTest ( <i>Draft of October 29, 2004</i> )	check for attribute nodes
AttributeValue	AttributeValue	define value of an attribute in construction
AttributeValueContent	AttributeValueContent	define value content of an attribute in construction
Axis	ForwardAxis, ReverseAxis	direction of movement on a path
AxisStep	ForwardStep, ReverseStep	create and possibly filter a sequence of nodes
CommentTest	CommentTest	check for comment nodes

Tamino XQuery Syntax Construct	W3C XQuery Expression	Short Description
CompCommentConstructor	CompCommentConstructor (Draft of October 29, 2004)	construct an XML comment by computing an expression
CompExpr	CompExpr (Draft of October 29, 2004)	comparison expression
CompPIConstructor	CompPIConstructor (Draft of October 29, 2004)	construct a processing instruction by computing an expression
CompTextConstructor	CompTextConstructor (Draft of October 29, 2004)	construct a text node by computing an expression
ComputedAttributeConstructor	ComputedAttributeConstructor	construct an attribute by computing an expression
ComputedDocumentConstructor	CompDocConstructor	construct a document by computing an expression
ComputedElementConstructor	ComputedElementConstructor	construct an element by computing an expression
Constructor	Constructor	constructor expression
DefaultCollationDecl	DefaultCollationDecl (Draft of November 12, 2003)	declare a default collation
DefaultNamespaceDecl	DefaultNamespaceDecl (Draft of November 12, 2003)	declare a default (function) namespace
DeleteClause	—	delete a node sequence
DirectCommentConstructor	DirCommentConstructor (Draft of October 29, 2004)	construct an XML comment
ElementConstructor	ElementConstructor	construct an XML element
ElementContent	ElementContent	define content of an element in construction
ElementNameOrFunctionCall	FunctionCall	represent an element or call a function
ElementTest	ElementTest (Draft of October 29, 2004)	check for element nodes
EnclosedExpr	EnclosedExpr	evaluate a sequence of expressions
Expr	Expr	basic XQuery expression
ExprSequence	ExprSequence	construct a sequence of expressions
FLWORExpr	FLWORExpr (Draft of October 29, 2004)	iterate over sequences of items
FLWUExpr	—	perform one or more update operations
ForClause	ForClause	bind variables by evaluating expressions



Tamino XQuery Syntax Construct	W3C XQuery Expression	Short Description
<a href="#">FunctionDecl</a>	<a href="#">FunctionDecl</a> ( <i>Draft of October 29, 2004</i> )	declare a user-defined function
<a href="#">GeneralComp</a>	<a href="#">GeneralComp</a>	compare sequences
<a href="#">GeneralStep</a>	subsumed in <a href="#">StepExpr</a>	create and possibly filter item sequence
<a href="#">IfExpr</a>	<a href="#">IfExpr</a> ( <i>Draft of October 29, 2004</i> )	conditional expression based on <code>if</code> , <code>then</code> , and <code>else</code>
<a href="#">InsertClause</a>	—	insert a node sequence at update position
<a href="#">IntersectExceptExpr</a>	<a href="#">IntersectExceptExpr</a>	combine node sequences
<a href="#">ItemType</a>	<a href="#">ItemType</a> ( <i>Draft of October 29, 2004</i> )	specify the type of an item
<a href="#">KindTest</a>	<a href="#">KindTest</a>	check the kind of node
<a href="#">LetClause</a>	<a href="#">LetClause</a>	bind variables by evaluating expressions
<a href="#">LibraryModule</a>	<a href="#">LibraryModule</a> ( <i>Draft of October 29, 2004</i> )	provide XQuery fragment to other modules
<a href="#">Literal</a>	<a href="#">Literal</a>	numeric or string literal
<a href="#">MainModule</a>	<a href="#">MainModule</a> ( <i>Draft of October 29, 2004</i> )	obligatory XQuery module with query body
<a href="#">ModuleDecl</a>	<a href="#">ModuleDecl</a> ( <i>Draft of October 29, 2004</i> )	identify library module
<a href="#">ModuleImport</a>	<a href="#">ModuleImport</a> ( <i>Draft of October 29, 2004</i> )	import one or more library modules
<a href="#">MultiplicativeExpr</a>	<a href="#">MultiplicativeExpr</a>	multiply or divide numerical value
<a href="#">NamespaceDecl</a>	<a href="#">NamespaceDecl</a> ( <i>Draft of November 12, 2003</i> )	declare a namespace
<a href="#">NameTest</a>	<a href="#">NameTest</a>	select nodes by name or wildcard
<a href="#">NoAxisStep</a>	—	create and possibly filter node sequence on current axis
<a href="#">NodeComp</a>	<a href="#">NodeComp</a> ( <i>Draft of October 29, 2004</i> )	compare two nodes
<a href="#">NodeTest</a>	<a href="#">NodeTest</a>	check if nodes satisfy conditions on the kind or name
<a href="#">NumericLiteral</a>	<a href="#">NumericLiteral</a>	numeric literal
<a href="#">OrderByClause</a>	<a href="#">OrderByClause</a> ( <i>Draft of October 29, 2004</i> )	order intermediate results
<a href="#">OrExpr</a>	<a href="#">OrExpr</a>	check if one or both of two expression are logically true

Tamino XQuery Syntax Construct	W3C XQuery Expression	Short Description
ParenthesizedExpr	ParenthesizedExpr	expression sequence enclosed in parentheses
PathExpr	PathExpr	select nodes on a path
Pragma	Pragma <i>(Draft of October 29, 2004)</i>	query processor directive
PrimaryExpr	PrimaryExpr	basic XQuery primitive
ProcessingInstructionTest	ProcessingInstructionTest	check for processing instruction nodes
Prolog	Prolog <i>(Draft of October 29, 2004)</i>	create environment for query processing
RangeExpr	RangeExpr	construct a sequence of consecutive integers
RelativePathExpr	RelativePathExpr	select nodes on the relative path
RenameClause	—	rename the nodes of a node sequence
ReplaceClause	—	replace a node
SequenceType	SequenceType <i>(Draft of October 29, 2004)</i>	specify the type of an XQuery value
SerializationSpec	XSLT 2.0 and XQuery 1.0 Serialization <i>(Draft of November 12, 2003)</i>	specify serialization of query output
SortExpr	SortExpr	sort a sequence of items
SortSpecList	SortSpecList, SortModifier	define ordering expression and sort direction
StepExpr	StepExpr	create and possibly filter a sequence of items
StepQualifiers	Predicates	restrict item sequence to those matching one or more predicates
TaminoQPIExecution	—	determine parameters of query execution
TaminoQPIExplain	—	retrieve query execution plan
TaminoQPIInline	—	use default inlining strategy for user-defined functions
TaminoQPIOptimization	—	choose query optimization strategy
TextTest	TextTest	check for text nodes
TreatExpr	TreatExpr <i>(Draft of October 29, 2004)</i>	modify static type of operand
UnaryExpr	UnaryExpr	negate numerical value of an expression
UnionExpr	UnionExpr	combine node sequences

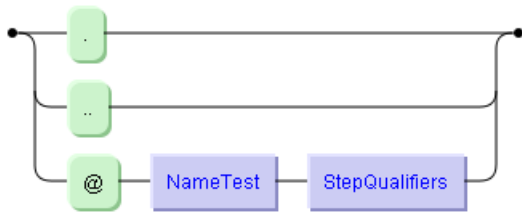
Tamino XQuery Syntax Construct	W3C XQuery Expression	Short Description
<a href="#">UpdateExpr</a>	—	update expression
<a href="#">UpdateIfExpr</a>	—	conditional update expression
<a href="#">UpdateSequence</a>	—	sequence of update expressions
<a href="#">UpdatingFunction</a>	—	updating function invocation
<a href="#">UpdatingFunctionDecl</a>	—	declaration of an updating function
<a href="#">ValueComp</a>	<a href="#">ValueComp</a>	compare single values
<a href="#">VarDecl</a>	<a href="#">VarDecl</a> ( <i>Draft of October 29, 2004</i> )	declare a variable in query prolog
<a href="#">VersionDecl</a>	—	specifies the version of the XQuery language — reserved for future use
<a href="#">WhereClause</a>	<a href="#">WhereClause</a>	filter intermediate results
<a href="#">Wildcard</a>	<a href="#">Wildcard</a>	select nodes matching the principal node kind
<a href="#">XmlProcessingInstruction</a>	<a href="#">DirPIConstructor</a>	construct an XML processing instruction
<a href="#">XQueryModule</a>	<a href="#">Module</a> ( <i>Draft of October 29, 2004</i> )	top-level XQuery syntax construct

## AbbreviatedStep

Select context, parent, or attribute node using abbreviated syntax.

### Syntax

---



### AbbreviatedStep

## Description

---

An `AbbreviatedStep` selects the context node, its parent node or some attribute node by using abbreviated syntax. The symbol `..` is short for `parent::node()`. The symbol `.` is short for `self::node()` if it is prepended by a path expression, otherwise it is short for the context item. Similarly, `@` refers to the attribute axis and is the abbreviation for `attribute::`. It is followed by a name test and zero or more predicate expressions, since the corresponding rule (`StepQualifiers`) allows an empty predicate. The `child` axis is the default axis, so that you can simply omit the axis name in front of the element name

## Examples

---

- Select all books published after 1990.

```
input()/bib/book[./@year > 1990]
```

This is short for `input()/child::bib/child::book[self::node()/attribute::year > 1990]`. Using `"."` to select the context node is not necessary here, but merely emphasizes that it is the element node `book`, where an attribute satisfying the predicate is looked for.

- Select all nodes that are parent of an element node `name`.

```
input()//name/..
```

This is short for `input()/descendant-or-self::node/child::name/parent::node()`. For the patient database this retrieves all `patient`, `nextofkin` and `doctor` nodes, since they all have `name` as an element child node.

- Select all doctors with a pager.

```
input()//doctor[@pager]
```

This is short for `input()/descendant-or-self::node/child::doctor[attribute::pager]`. The abbreviated step uses `@` to select the attribute axis and then checks for an attribute with the name `pager` () and selects all `doctor` elements that have an attribute node with the name `pager` (`NameTest` with empty `StepQualifiers`).

- Select all attribute nodes of all doctypes in the current collection.

```
input()//@*
```

Here, the name test following the `"@"` is a wildcard expression matching all of the selected kind of node.

- Sort a sequence of numbers:

```
(3, 1, 2) sort by (.)
```

In this example `.` selects the context item.

## Related Syntax Constructs

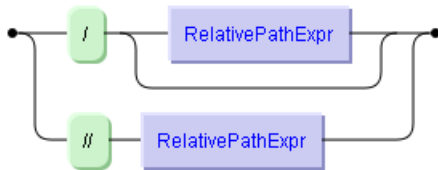
[NameTest](#)   [StepQualifiers](#)

## AbsolutePathExpr

Select nodes on the absolute path.

### Syntax

---



**AbsolutePathExpr**

### Description

---

An `AbsolutePathExpr` is a path expression originating in the document root `/`. There are two variants depending whether the path directly begins at the document node using `/`, or whether there is an initial sequence of nodes in front of the relative path expression that follows, which is the case for `//`.

An initial `/` in an `AbsolutePathExpr` is short for the step `fn:root(self::node())`. The path begins with the root node that contains the context node. An initial `//` is an abbreviation for the steps `fn:root(self::node())/descendant-or-self::node()`. A node sequence is created that contains all the nodes in the same hierarchy as the context node. For both variants of this expression it is an error if the context item is not a node.

### Related Syntax Construct

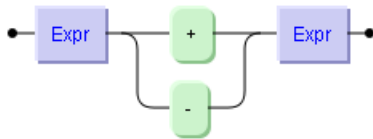
---

[RelativePathExpr](#)

## AdditiveExpr

Add or subtract numerical values.

## Syntax



### AdditiveExpr

## Description

The `AdditiveExpr` provides the usual arithmetic operations for adding and subtracting numerical values. Each operand is atomized so that it is either a single atomic value or an empty sequence. If one of the operands is atomized to an empty sequence, the result of the operation is also an empty sequence.

## Examples

- A sequence of expressions that consists of string and numeric literals:

```
(59 - 17)
```

As integer literals the operands are already atomic values and of valid type for the operator. Tamino returns the expected result 42.

- Compute the years, in which all patients turned 18 years:

```
for $a in input()/patient/born
return $a + 18
```

The variable `$a` is bound to the `born` element. After atomization the addition can be performed.

## AndExpr

Check if both of two expressions are logically true.

## Syntax

---



### AndExpr

## Description

---

An `AndExpr` is a logical expression that is evaluated by determining the effective boolean value (ebv) of each of its operands. The value of the `AndExpr` is then determined as follows:

and expression	ebv(op2) = true	ebv(op2) = false	error in ebv(op2)
ebv(op1) = true	true	false	error
ebv(op1) = false	false	false	false or error
error in ebv(op1)	error	false or error	error

## Example

---

- Select all patients born in 1950 and living in Bradford:

```
for $a in input()/patient
where $a/born = 1950 and $a//city = "Bradford"
return $a
```



## AnyKindTest

Check for any kind of node.

## Syntax

---



**AnyKindTest**

## Description

---

An `AnyKindTest` is one of a number of node tests that are used in a step expression. The test `node()` is true for any kind of node.

## Example

---

- Select all nodes:

```
input()/node()
```

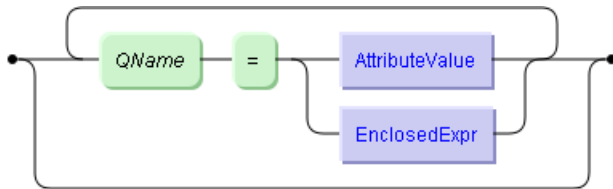
This test is true for all nodes found in the documents of the current collection. More precisely, since this query expression is equivalent to `input()/child::node`, it returns the sequence of children of document root nodes, i.e., root element nodes, comment nodes and processing instruction nodes.

## AttributeList

Creates a list of attributes.

## Syntax

---



**AttributeList**

## Description

---

An `AttributeList` is part of an `ElementConstructor` and creates a list of zero or more attributes. Each attribute is defined by a `QName` followed by an equals sign and either an enclosed expression or an `AttributeValue` that consists of literals or in turn an enclosed expression.

## Example

---

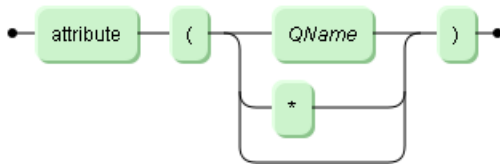
- Construct a section element with two attributes `type` and `lang`:

```
<section type="overview" lang="en">
  <para>This section deals with element constructors.</para>
</section>
```

## AttributeTest

Checks for attribute nodes.

## Syntax



**AttributeTest**

## Description

An `AttributeTest` checks for an attribute node and its name and can be of one of these forms:

- `attribute()` and `attribute(*)` match any single attribute node, regardless of its name or type.
- `attribute(QName)` matches any attribute node whose name is `QName`, regardless of its type.

## Example

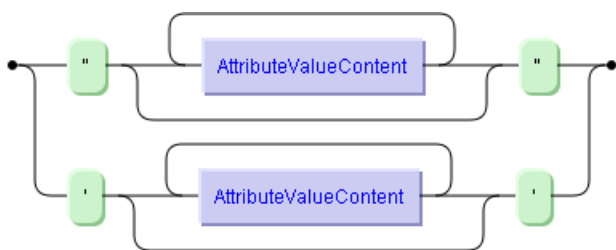
- The attribute test `attribute(id)` matches any attribute node whose name is "id".

## AttributeValue

Define value of an attribute in construction.

### Syntax

---



**AttributeValue**

### Description

---

An `AttributeValue` is part of an `AttributeList` inside an `ElementConstructor`. This expression specifies that attribute values are either delimited by single quotes or by double quotes.

### Example

---

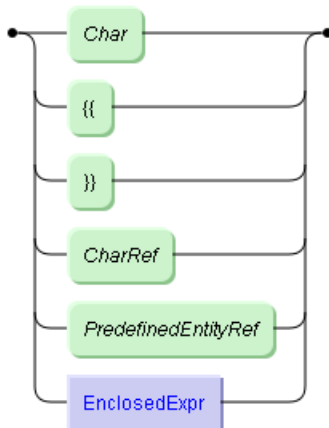
- Some string literals as attribute values:

```
<example quote1="This is Joe's house."
        quote2='I bought a 21" monitor today.'
        quote3="He's 5'11&quot; tall."/>
```

## AttributeValueContent

Define value content of an attribute in construction.

### Syntax



**AttributeValueContent**

### Description

This expression is part of an `AttributeValue`. It defines the content of an attribute value when using a constructor expression. As literals it allows any defined character, a character reference such as `&#x05D0;`, a predefined entity references such as `&quot;`, and two escape sequences for the left and right brace. Since the braces are used to delimit enclosed expressions, an escape sequence is necessary to use the braces as literals. For that purpose, two adjacent braces are interpreted as a single literal brace.

Furthermore, content can be determined by an enclosed expression that contains an expression which is evaluated and the result placed as element content.

### Example

See [AttributeValue](#) for examples.

## Related Syntax Constructs

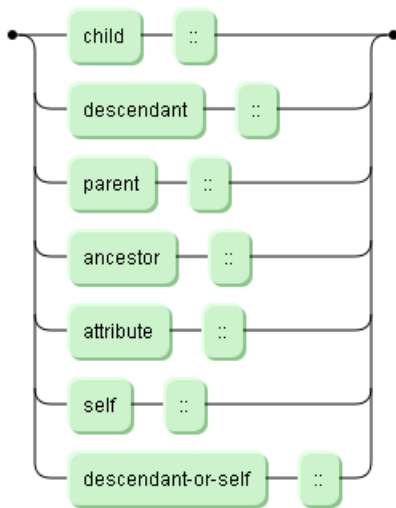
---

[AttributeList](#)

## Axis

Direction of movement on a path.

## Syntax



## Axis

## Description

An **Axis** determines the direction of movement, when a step on a path is taken, beginning at the context node. In the reverse direction, you can address the following axis:

```
AxisParent    parent::
AxisAncestor  ancestor::
```

In the forward direction, you can address the following axes:

```
AxisChild      child::
AxisDescendant descendant::
AxisAttribute   attribute::
AxisSelf        self::
AxisDescendantOrSelf descendant-or-self::
```

For all axes except the **ancestor** and **descendant** axes there is an abbreviated syntax. See [AbbreviatedStep](#) for details.

## Example

---

- Select all books published in the 21st century:

```
input()/descendant::book[attribute::year > 2000 ]
```

Starting from the document node, this query looks for any descendant `book` elements. In the predicate expression the attribute axis is searched for an attribute `year` whose numerical value is larger than 2000.

## Related Syntax Constructs

---

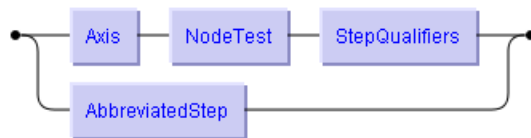
[AbbreviatedStep](#)   [NodeTest](#)   [StepExpr](#)   [StepQualifiers](#)



## AxisStep

Create and possibly filter a sequence of nodes.

### Syntax



#### AxisStep

### Description

An `AxisStep` is a step expression that creates a sequence of nodes. Beginning at the context node, it takes a step in forward or reverse direction of the given axis.

The unabbreviated form requires a node test that checks whether the node is of a certain kind or whether it passes a name test. You can specify a predicate expression by using `StepQualifiers` to filter the current sequence of nodes.

The abbreviated form allows you to use shorthand notations for the axes `self`, `parent` and `attribute`.

### Example

- Select all books written by Stevens:

```
input()/descendant-or-self::book[child::author[child::last = "Stevens"]]
```

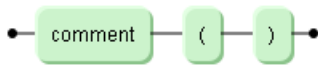
Starting from the document node, this query uses the unabbreviated form to take a step on the `descendant-or-self` axis and adds any `book` (`NodeTest`) to the sequence. Those elements are retained that satisfy the following predicate expression (`StepQualifiers`): there is a child element node `author` that has a child element node `last` whose string value is "Stevens".

## CommentTest

Check for comment nodes.

## Syntax

---



### CommentTest

## Description

---

A `CommentTest` is one of a number of node tests that are used in a step expression. The expression `comment()` is **true** for any comment node.

## Example

---

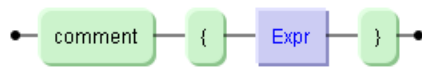
- Select all comment nodes in the current collection:

```
input()//comment()
```

## CompCommentConstructor

Construct an XML comment by computing an expression.

### Syntax



### CompCommentConstructor

### Description

A `CompCommentConstructor` constructs an XML comment by computing the enclosed `Expr`. That expression is evaluated and, after atomization, converted into a sequence of atomic values each of which is cast into a string and concatenated with a space character between each pair of values. It is an error if the result of the expression after atomization contains two adjacent hyphens or if it ends with a hyphen.

### Example

- This query constructs the XML comment `<!--Houston, we have a problem.-->`:

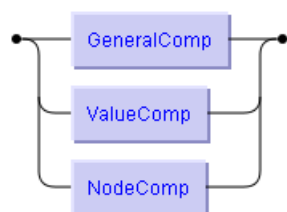
```
let $homebase := "Houston"
return comment { string-join(($homebase, ", we have a problem."), "") }
```

## CompExpr

Comparison expression.

## Syntax

---



**CompExpr**

## Description

---

A `CompExpr` compares two values by applying one of the following comparison expressions:

- A `GeneralComp` is an existentially quantified comparison.
- A `NodeComp` compares two nodes by their identity or by their document order.
- A `ValueComp` compares two single values.

## Related Syntax Construct

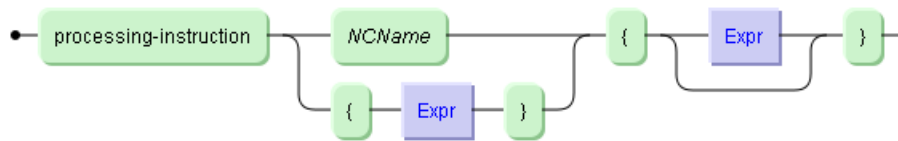
---

[ElementConstructor](#)

## CompPIConstructor

Construct a processing instruction by computing an expression.

### Syntax



### CompPIConstructor

### Description

A `CompPIConstructor` constructs a processing instruction (PI) by computing the enclosed `Expr`, called *content expression*. The name of the element is either a `NCName` or it is computed from an expression (*name expression*).

The value resulting from evaluating the name expression undergoes atomization, and must be a single atomic value of the type `xs:NCName`, `xs:string`, or `xdt:untypedAtomic`, otherwise a type error is raised. If necessary, the value is cast into type `xs:NCName`. It is then used as the PI target.

The content expression is evaluated and, after atomization, converted into a sequence of atomic values each of which is cast into a string and concatenated with a space character between each pair of values. If the result of the expression after atomization is an empty sequence, it is replaced by a string of length zero.

### Example

- This constructs the processing instruction `<?audio-output beep?>`:

```

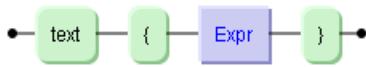
let $target := "audio-output",
    $content := "beep"
return processing-instruction {$target} {$content}
  
```

## CompTextConstructor

Construct a text node by computing an expression.

### Syntax

---



#### CompTextConstructor

### Description

---

A `CompTextConstructor` constructs a text node by computing the enclosed expression. This expression is evaluated and, after atomization, converted into a sequence of atomic values each of which is cast into a string and concatenated with a space character between each pair of values. If the result of the expression after atomization is an empty sequence, no text node is constructed.

### Example

---

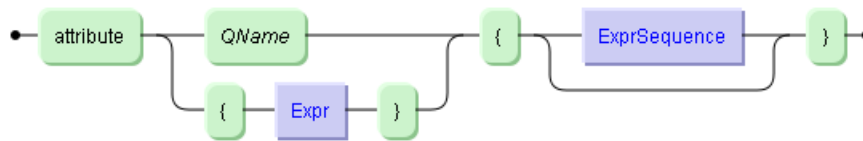
- This is the XQuery version of the classic example of printing "Hello World":

```
text{"Hello World"}
```

## ComputedAttributeConstructor

Construct an attribute by computing an expression.

### Syntax



### ComputedAttributeConstructor

### Description

A `ComputedAttributeConstructor` constructs an attribute by computing the enclosed `ExprSequence`, called *content expression*. The name of the attribute is either a `QName` or it is computed from an expression (*name expression*).

The value resulting from evaluating the name expression undergoes atomization, and must be a single atomic value of the type `xs:QName`, `xs:string`, or `xdt:untypedAtomic`, otherwise a type error is raised. If it is of type `xs:QName`, its expanded `QName` is used as element name and the prefix part of the `QName` is retained. Otherwise, the value is converted to an expanded `QName`. If a namespace prefix is present, it is resolved using the statically known namespaces, otherwise it is local in the default namespace.

The content expression is evaluated and, after atomization, converted into a sequence of atomic values each of which is cast into a string and concatenated with a space character between each pair of values. If the result of the expression after atomization is an empty sequence, the string is of length zero.



**Tip:** A computed attribute constructor is needed in the context of update expressions, since you cannot construct a stand-alone attribute with the constructors using regular XML syntax.

### Example

- This query constructs an attribute node named `size` and the computed value "6":

```
attribute size { 4 + 2 }
```

- Computed element constructors may be nested and several constructors are separated by commas:

```
element section {  
  attribute type { "overview" },  
  attribute lang { "en" },  
  element para { "This section deals with attribute constructors." }  
}
```

See the section *Performing Update Operations in the XQuery 4 User Guide* for information about updating attribute nodes with the help of a `ComputedAttributeConstructor`.



## ComputedDocumentConstructor

A `ComputedDocumentConstructor` constructs a document node with a new node identity. The result of evaluating the `ExprSequence` yields the child nodes of the constructed document node.

### Syntax



### Description

When creating an XML document node, the result of the evaluation of the `ExprSequence` must be well-formed XML, i.e.:

- it may contain comment or processing instruction nodes;
- it must contain exactly one element node;
- it must not contain any attribute nodes or text nodes.

Any XML document nodes contained in that sequence are implicitly replaced by their child nodes. For XML document construction, the sequence must not contain any simple type values.

In Tamino XQuery, a document constructor can also be used to create a non-XML document node. In this case the following applies to the result of evaluating the `ExprSequence`:

- It may be empty. This will result in an empty non-XML document.
- It may be a non-XML document node. This results in a non-XML document node with the same content (but a new node identity). Also the content-type is inherited from the source document.
- It may be a string value. This creates a non-XML text document with the string as its text content.

The `ExprSequence` must not result in both XML and non-XML content.

For creating documents with extra options (content-type, ACL), see the function `tf:document()`.

### Examples

- This query creates an XML document node of a book document:

```
document { <book><title></title></book> }
```

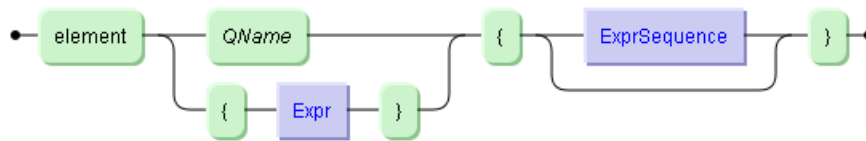
- This query contains a non-XML document holding the source text of an XQuery module:

```
document {"module namespace m='M'; declare function m:f() {'F'};"}
```

## ComputedElementConstructor

Construct an element by computing an expression.

### Syntax



### ComputedElementConstructor

### Description

A `ComputedElementConstructor` constructs an element by computing the enclosed `ExprSequence`, called *content expression*. The name of the element is either a `QName` or it is computed from an expression (*name expression*).

The value resulting from evaluating the name expression undergoes atomization, and must be a single atomic value of the type `xs:QName`, `xs:string`, or `xdt:untypedAtomic`, otherwise a type error is raised. If it is of type `xs:QName`, its expanded `QName` is used as element name and the prefix part of the `QName` is retained. Otherwise, the value is converted to an expanded `QName`. If a namespace prefix is present, it is resolved using the statically known namespaces, otherwise it is local in the default namespace.

The content expression is evaluated in the same way as an enclosed expression in an `ElementConstructor`.

### Examples

- Computed constructors may be nested and several constructors are separated by commas. Note how the expression sequence is evaluated for the nested element constructors:

```
for $a in input()/bib/book
return
element book {
  element title { $a/title },
  element notitle { "$a/title" }
}
```

This query is equivalent with the following one:

```
for $a in input()/bib/book
return
<book>
  <title>{ $a/title }</title>
  <notitle>{ "$a/title" }</notitle>
</book>
```

- This query returns the alphabetical list of patient names grouped by the initial letter of the patients' surnames:

```
let $patients := input()/patient/name
let $letters := distinct-values(for $patient in $patients
                                return substring($patient/surname, 1, 1))
return
  <patients> {
    for $letter in $letters
    return element { $letter } {
      for $patient in $patients[substring(surname, 1, 1) = $letter]
      order by $patient/surname
      return $patient }
  }
</patients>
```

Here, the name of the element is computed using the sequence of unique letters resulting from the first nested FLWOR expression. The content expression is another FLWOR expression returning the ordered list of all `name` elements for all letters found.

## Related Syntax Constructs

---

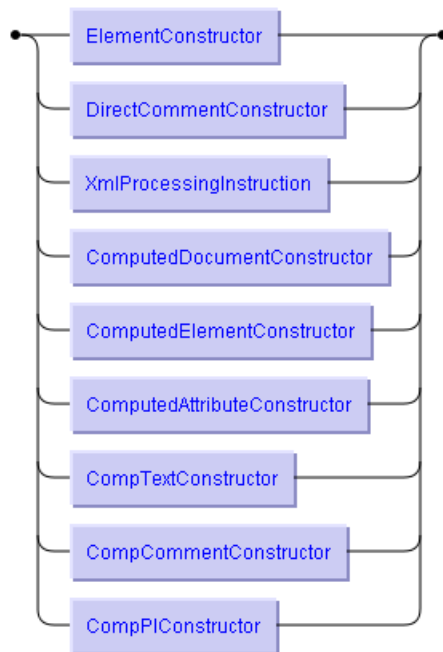
[ElementConstructor](#)

## Constructor

Constructor expression.

## Syntax

---



## Constructor

## Description

---

A `Constructor` is a general expression that is defined by further expressions to construct different kinds of nodes.

## Example

---

See the description of the respective constructor expressions for examples.

## DefaultCollationDecl

Declare a default collation.

### Syntax

---



#### DefaultCollationDecl

### Description

---

A `DefaultCollationDecl` is part of the query prolog and declares a default collation for later use in the query body. It consists of the keywords `declare default collation`, and a string literal, which is the collation URI.

The collation URI consists of a string literal which is interpreted as relative URL with the base URI `http://www.softwareag.com/tamino`. It is composed of `collation?<option>=<value>`. The available options correspond to the child elements of the TSD schema element `tsd:collation`. They are:

Collation Option	Value
<code>alternate</code>	specifies how to handle punctuation
<code>caseFirst</code>	specifies how to treat the first character of a word
<code>caseLevel</code>	use an extra comparison level for case differences (boolean value)
<code>french</code>	use French accent sorting (boolean value)
<code>language</code>	language code according to ISO 639 (see the list of language codes available in Tamino for details)
<code>normalization</code>	perform text normalization (boolean value)
<code>strength</code>	one of five comparison levels

The schema user guide provides detailed information about these collation options. See also the Tamino XQuery function [tf:getCollation](#).

### Example

---

- Declare default collation for the French language:

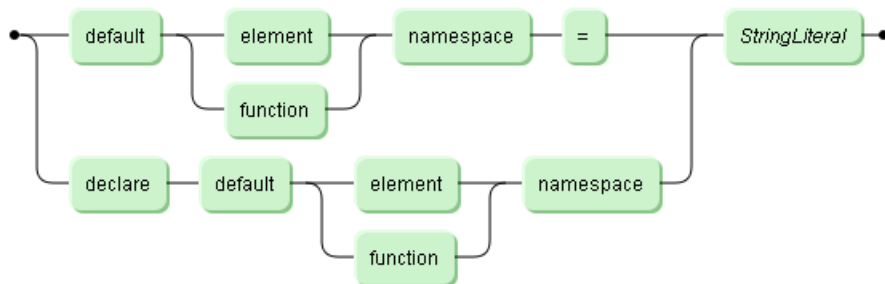
```
declare default collation "collation?language=fr"
```

## DefaultNamespaceDecl

Declare a default namespace.

### Syntax

---



### DefaultNamespaceDecl

### Description

---

A `DefaultNamespaceDecl` is part of a prolog and declares a default namespace in order to use unprefixed `QNames`. It consists of the keyword sequence `declare default`, followed by either of the keywords `element` or `function`, the keyword `namespace`, and a string literal representing the namespace URI.

As the syntax suggests, you can declare a default namespace for elements as well as functions. However, a prolog must not contain more than one of each kind of declaration. If no default function namespace is declared, it is the namespace of the XQuery/XPath functions, *http://www.w3.org/2002/08/xquery-functions*, which is prefixed by "fn" and "xf".



**Note:** The syntax variant without the keyword `declare`, but with the equals sign, was used in previous versions of the W3C XQuery specification as well as in Tamino. It has been retained for compatibility reasons.

### Examples

---

- Declare a default namespace for mathematical functions:



```
declare default function namespace "http://www.examples.com/math-functions"
```

- Declare company-internal namespace for elements:

```
declare default element namespace "http://company.dot.com/namespaces/corporate"
```

## Related Syntax Construct

---

[NamespaceDecl](#)

## DeleteClause

Delete a node sequence.

## Syntax

---



### DeleteClause

## Description

---

A `DeleteClause` is part of an `UpdateExpr` and deletes the node sequence that is the result of the evaluation of the expression following the keyword `delete`.

## Examples

---

- Strip all comments in the documents of the current collection:

```
update delete input()//comment()
```

- Delete all books that do not have a publishing year:

```
update delete input()/bib/book[not(@year)]
```

For the sample data provided, `year` is a required attribute of the element `book`, so the expression evaluates to an empty sequence and none of the `book` elements are deleted.

- Clear all element nodes in the current collection:

```
update delete input()//*/text()
```

The expression evaluates to all element nodes. Their text contents are deleted, but neither the elements themselves nor their attributes are deleted.

## DirectCommentConstructor

Construct an XML comment.

### Syntax



### DirectCommentConstructor

### Description

A `DirectCommentConstructor` creates a comment node. Its parent node is the node constructed by the nearest containing element or document node constructor, either direct or computed, if such a constructor exists; otherwise the parent node is empty.

An XML comment begins with "`<!--`" and ends with "`-->`". In between it may not contain two consecutive hyphens and it may not end with a hyphen. Formally, this is:

```
"<!--" ((Char - '-' | ('-' (Char - '-')))* "-->"
```



**Note:** The XML comment created by this constructor is different from an XQuery comment.

### Example

- This is a directly constructed comment:

```
<!-- Page Header -->
```

### Related Syntax Construct

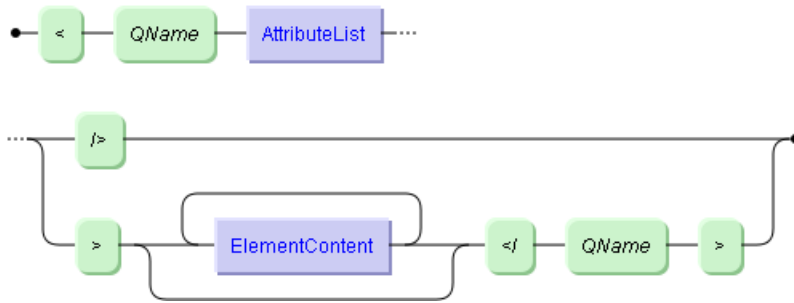
[ElementConstructor](#)

## ElementConstructor

Construct an XML element.

### Syntax

---



### ElementConstructor

### Description

---

An `ElementConstructor` constructs an XML element. It consists of a `QName`, that is used in start and end tag, an attribute list that may be empty, and the actual content of the element that may be empty as well.

You can construct elements in the following ways:

#### ■ Using Direct Constructors

If name, attributes and content of the element are constants, the element is constructed as is:

```
<section type="overview" lang="en">
  <para>CVS marks version conflicts by displaying the
    conflicting areas from both files and delimits this
    section using <![CDATA[<<<<<<]]> at the beginning
    and <![CDATA[>>>>>>]]> at the end.</para>
</section>
```

This constructs the element `section` with an attribute list containing the attributes `type` and `lang`. In terms of the data model, it constructs an element node and two attribute nodes. The content is another element `para` which in turn is constructed by an element constructor. That element has no attribute, but character data and two enclosed CDATA sections. For `para` the constructor creates an element node and a non-empty text node.

### ■ Using Enclosed Expressions

Enclosed expressions are delimited by braces. They are evaluated and replaced with the resulting value:

```
for $a in input()/bib/book
return
<book>
  <title>{ $a/title }</title>
  <notitle>$a/title</notitle>
</book>
```

The expression { \$a/title } is evaluated and for each tuple returned by the `for` clause it is replaced with the contents of the respective `title`. The content of the `notitle` element is a constant string literal and will be constructed as is.

### ■ Using Computed Constructors

Tamino also supports computed constructors for elements and attributes. They are introduced with either the keyword `element` or the keyword `attribute` followed by the name of the node, which must be a string literal, and an enclosed expression that evaluates to the content of the respective node. Computed constructors may be nested and several constructors are separated by commas:

```
element section {
  attribute type { "overview" },
  attribute lang { "en" },
  element para { "This section deals with element constructors." }
}
```

This constructs the same element as in the first example. Similarly, you can construct the second example:

```
for $a in input()/bib/book
return
element book {
  element title { $a/title },
  element notitle { "$a/title" }
}
```

This form of attribute constructor in the first expression is needed in the context of update expressions. See the section *Performing Update Operations* in the *XQuery 4 User Guide* for information about updating attribute nodes with the help of a `ComputedAttributeConstructor`.

## Related Syntax Constructs

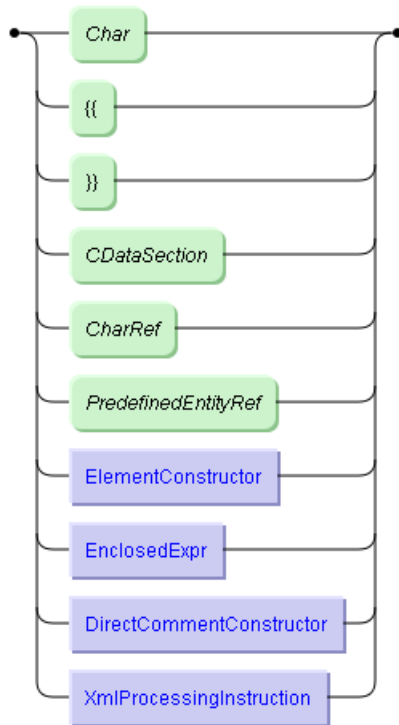
[AttributeList](#)

## ElementContent

Define content of an element in construction.

### Syntax

---



### ElementContent

### Description

---

An `ElementContent` is part of an `ElementConstructor`. It defines the content of an element if you use an element constructor. As literals it allows any defined character, a character reference such as `&#x05D0;`, a predefined entity references such as `&quot;`, and two escape sequences for the left and right brace. Since the braces are used to delimit enclosed expressions, an escape sequence is necessary to use the braces as literals. For that purpose, two adjacent braces are interpreted as a single literal brace.

Furthermore, content can be an element constructor enabling nested element structures, direct constructors creating CDATA sections, XML comments or processing instructions, and enclosed expressions that contain an expression which is evaluated and the result placed as element content.

## Example

---

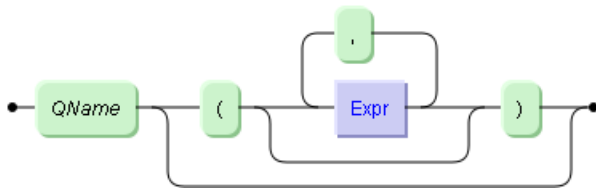
See [ElementConstructor](#) and [EnclosedExpr](#) for examples.

## ElementNameOrFunctionCall

Represent an element or call a function.

### Syntax

---



`ElementNameOrFunctionCall`

### Description

---

An `ElementNameOrFunctionCall` consists of a `QName` that may be followed by a list of comma-separated expressions enclosed in parentheses. If it only consists of a `QName`, then it represents an element name. Otherwise it is a function call.

In case of a function call, each argument is evaluated. The resulting value is converted to the declared type of the corresponding function parameter as described in the section *Data Types* of the *XQuery 4 User Guide*. The function is then executed using the converted argument values. The return value is converted to the type declared as the return type of that function.

### Example

---

- Compute the number of books published before 2000:

```
count(for $a in input()/bib/book
      where $a/@year < 2000
      return $a)
```

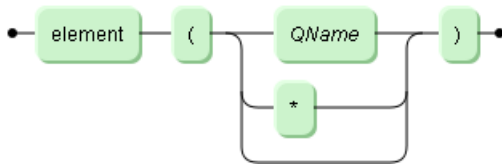
The `count` function takes a sequence of items, and returns the number of items in the sequence. Here, the sequence of all `book` elements that satisfy the condition that they have an attribute `year` whose value is smaller than 2000, is passed as argument to the `count` function. The computed number of sequence items is then returned as an integer value.



## ElementTest

Check for element nodes.

## Syntax



ElementTest

## Description

An `ElementTest` checks for an element node and its name and can be of one of these forms:

- `element()` and `element(*)` match any single element node, regardless of its name or type.
- `element(QName)` matches any element node whose name is `QName`, regardless of its type.

## Example

- The element test `element(book)` matches any element node whose name is "book".

## EnclosedExpr

Evaluate a sequence of expressions.

## Syntax

---



### EnclosedExpr

## Description

---

An `EnclosedExpr` is part of constructors for elements and attributes and consists of a sequence of one or more expressions that is enclosed by braces. The expressions are evaluated and the `EnclosedExpr` is replaced with the result of the evaluation.

## Example

---

- Reformat book entries:

```
for $a in input()/bib/book
let $authors :=
  for $I in $a/author
  return <author>{ $i/last/text() }, { $i/first/text() }</author>
return
<mybook published="{ $a/@year }">
  { $authors }
  { $a/title }
</mybook>
```

This example uses five enclosed expressions that are evaluated to three different kinds of nodes: two text nodes that are used in the construction of the local `author` elements bound to `$authors`, an attribute node (`year`), a single element node (`title`), and one or more element nodes by evaluating `$authors`.

## Related Syntax Construct

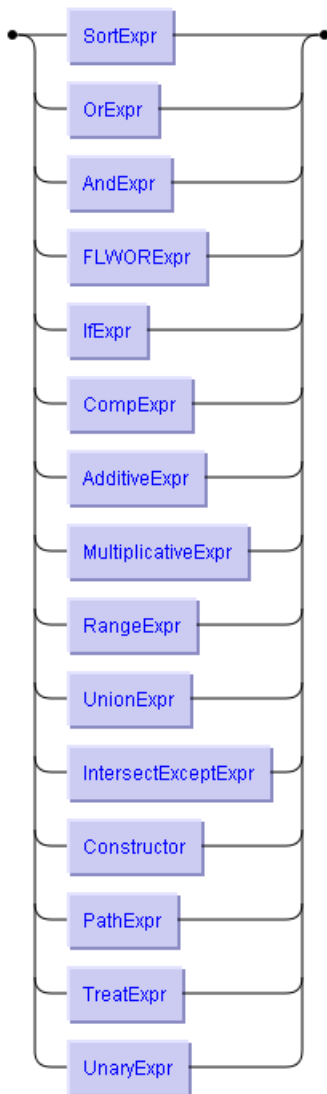
---

[ElementConstructor](#)

## Expr

Basic XQuery expression.

## Syntax



**Expr**

## Description

An `Expr` represents a basic XQuery expression and provides several kinds of expressions. Each kind of expression is defined in terms of other expressions (compositionality).

## Precedence Order

Operands with a higher precedence number bind more tightly than those with a lower precedence number. Operands listed at the same level are evaluated from left to right.

Precedence Number	Syntax Constructs
1 (lowest)	<a href="#">SortExpr</a>
2	<a href="#">OrExpr</a>
3	<a href="#">AndExpr</a>
4	<a href="#">CompExpr</a> , <a href="#">FLWORExpr</a> , <a href="#">IfExpr</a>
5	<a href="#">RangeExpr</a>
6	<a href="#">AdditiveExpr</a>
7	<a href="#">MultiplicativeExpr</a>
8	<a href="#">UnionExpr</a>
9	<a href="#">IntersectExceptExpr</a>
10	<a href="#">TreatExpr</a>
11	<a href="#">UnaryExpr</a>
12 (highest)	<a href="#">Constructor</a> , <a href="#">PathExpr</a>

## Example

---

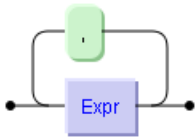
See the pages of the various expressions for examples.

## ExprSequence

Construct a sequence of expressions.

### Syntax

---



ExprSequence

### Description

---

An `ExprSequence` constructs a sequence of expressions that consists of at least one expression. Further expressions are separated by commas.

### Example

---

- A sequence of expressions that consists of string and numeric literals:

```
(1, 2, ("a", "b", "c"), 3, 4)
```

Since sequences cannot be nested, this sequence is evaluated to the sequence (1, 2, a, b, c, 3, 4).

### Related Syntax Constructs

---

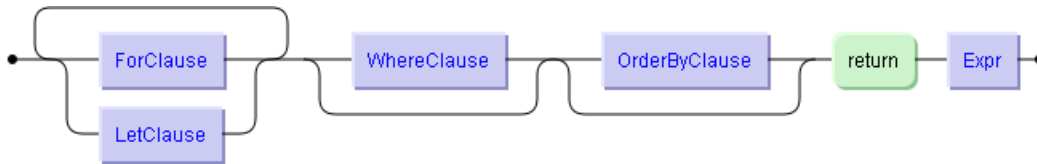
[XQueryModule](#)

## FLWORExpr

Iterate over sequences of items.

### Syntax

---



FLWORExpr

### Description

---

A FLWOR expression iterates over a sequence of items and binds variables that can be used in the scope of the current expression. If the item sequence is empty, the result of the FLWOR expression is an empty sequence. A FLWOR expression consists of one or more `for` and `let` clauses in any combination, followed by an optional `where` clause, an optional `order by` clause and a `return` clause. Briefly, these clauses are interpreted as follows:

- A `for` clause binds one or more variables to each value of the result of the following expression.
- A `let` clause binds one or more variables to the complete result of the expression.
- A `where` clause retains only those intermediate results that satisfy the following condition.
- An `order by` clause can reorder the intermediate results.
- A `return` clause evaluates the following expression and returns the result.

### Examples

---

- Retrieve a list of book titles along with a current list number:

```
for $a at $i in input()/bib/book
return (<BookNo>{ $i }</BookNo>, $a/title)
```

A positional variable is used for creating the current list number, see the description of the [ForClause](#) for details.

## ■ Join Operations

1. Since a `for` clause creates tuples of variable bindings from the Cartesian product of the sequences that the expressions evaluate to, a *cross join* is straightforward:

```
for $i in (1, 2), $j in (3, 4)
return
  <tuple>
    <i>{ $i }</i>
    <j>{ $j }</j>
  </tuple>
```

In SQL contexts, this concept corresponds to the `CROSS JOIN` expression.

2. Select all books for which a review exists, with all authors, title and the review text:

```
for    $b in input()/bib/book,
      $a in input()/reviews/entry
where  $b/title = $a/title
order by $b/title
return
  <book>
    { $b/author }
    { $b/title }
    { $a/review }
  </book>
```

This FLWOR expression is an example of an *equijoin*: from both doctypes `bib` and `reviews` only those tuples are retained that satisfy the equality condition in the `where` clause. The tuples are then sorted according to the book title in ascending order. See also the result in the *XQuery 4 User Guide*. In SQL contexts, this concept corresponds to the `JOIN ... ON` expression.

- A FLWOR expression can often simply be a replacement for a path expression using predicates (filter):

```
input()/bib/book[author/last = "Stevens"]
```

can also be written as:

```
for $a in input()/bib/book
where $a/author/last = "Stevens"
return $a
```

Although this query expression is longer than the first one, it generally performs better and it is recommended to use FLWOR expressions instead of path expressions with filters. See also the section *Efficient Queries: XQuery* in the *Performance Guide*.

## Related Syntax Constructs

---

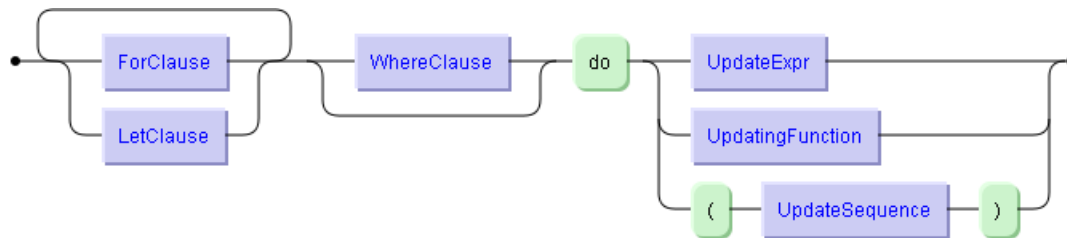
[ForClause](#)   [LetClause](#)   [OrderByClause](#)   [WhereClause](#)



## FLWUEExpr

Perform one or more update operations.

### Syntax



FLWUEExpr

### Description

A `FLWUEExpr` is a variant of the regular `FLWOR` expression. Like the other update expressions it is introduced by the keyword `update`. In contrast to a `FLWOR` expression it has no `return` clause, but a `do` clause that may contain a single basic update expression or more than one basic update expressions that are separated by commas and enclosed in parentheses.

For more information on how the clauses `for`, `let` and `where` are treated, please see the documentation to [FLWORExpr](#) in this reference guide or the section `FLWOR Expressions` in the *XQuery 4 User Guide*.

If you specify two or more update operations that act on the same update node, conflicts may arise. Depending on the type of conflict, the operation may or may not be rejected by Tamino. See the section `Conflicts` in the *XQuery 4 User Guide*.

### Example

- Delete all books in which Stevens appear as author or editor:

```
update for $a in input()/bib/book
where $a/author/last = "Stevens" or $a/editor/last = "Stevens"
do delete $a
```

## Related Syntax Constructs

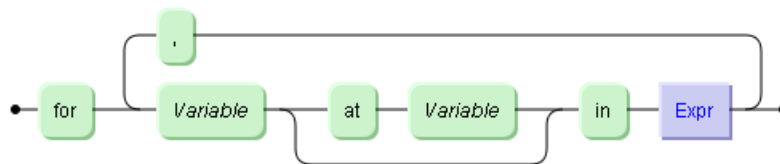
---

[FLWORExpr](#)

## ForClause

Bind variables by evaluating expressions.

## Syntax



ForClause

## Description

A `ForClause` is part of `FLWOR` and `FLWU` expressions and contains one or more variables that will be related to the expression that follows. The `for` clause creates tuples from the Cartesian product that is computed from the sequence of values which are the result of evaluating the expression. The `return` clause will be invoked once for each generated tuple that is retained after passing the optional `where` clause.

To each bound variable you can associate a *positional variable* that is bound at the same time. It is preceded by the keyword `at` and is always of the implied type `xs:integer`. The positional variable represents the ordinal position of the items in the binding sequence over which the variable iterates, starting at 1.

## Example

See [FLWORExpr](#) for examples.

## Related Syntax Constructs

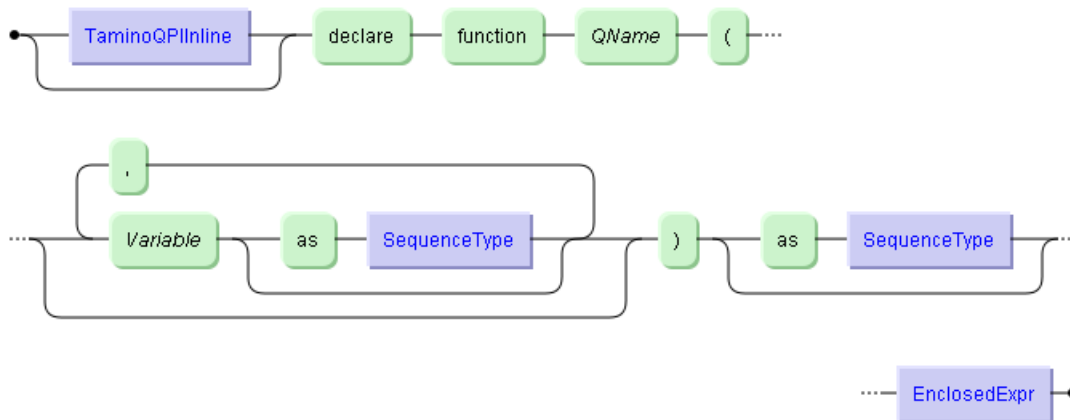
[LetClause](#)   [WhereClause](#)

## FunctionDecl

Declare a user-defined function.

### Syntax

---



### FunctionDecl

### Description

---

A `FunctionDecl` is part of a prolog. It declares a user-defined function by specifying its name, a parameter list and the optional result type. The enclosed expression is called the *function body*. The parameter list is enclosed in parentheses and comprises a comma-separated list of parameters with their names and optional types. The scope of the function parameters is the function body. A parameter name may appear only once in the function declaration.

The type information is optional: If a parameter is specified without a type then the default type `item*` is used. If the type of the result value is not specified, again `item*` is used. If you specify a type, you can use one of the following XQuery types as parameter or return types: all simple types, `item`, `node`, `attribute`, `comment`, `document`, `element`, `processing-instruction`, and `text`. This also means that you cannot use a user-defined type as return or parameter type.

Every function must be in a namespace, that is every declared function name must have a non-empty namespace URI. Every function name declared in a library module must be in the target namespace of the library module. However, in a main module you can declare functions for local use without defining a new namespace. For this purpose, the namespace preceded by the URI *local* is predeclared and bound to <http://www.w3.org/2004/07/xquery-local-functions>.

Function declarations may be recursive. Also, mutually recursive functions are allowed, that is, functions whose bodies reference each other.

Furthermore, you can prepend the query directive `TaminoQPIInline` to affect the way this function is optimized by the query processor.

## Example

---

- This recursive function returns the depth of the current node on the element axis.

```
module namespace tree="http://www.examples.com/tree"
declare function tree:depth($e as node()) as xs:integer
{
  if (not($e/*))
  then
    1
  else
    max(for $c in $e/* return tree:depth($c)) + 1
}
```

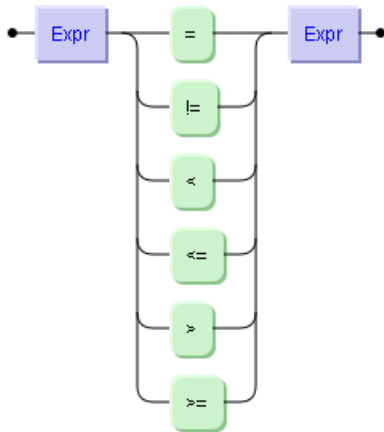
This function declaration is part of a library module. In order to use it directly in a main module instead, drop the module declaration and use the namespace prefix "local" instead of "tree".

## GeneralComp

Compare sequences.

## Syntax

---



### GeneralComp

## Description

---

A general comparison differs from a **value comparison** in that operands may be sequences of any length with implicit existential quantification. Given two sequences A and B the result is determined as follows:

- $A = B$  is true if and only if the value comparison  $a \text{ eq } b$  is true for some item  $a$  in A and some  $b$  in B. Otherwise it is false.
- $A \neq B$  is true if and only if the value comparison  $a \text{ ne } b$  is true for some item  $a$  in A and some  $b$  in B.
- $A < B$  is true if and only if the value comparison  $a \text{ lt } b$  is true for some item  $a$  in A and some  $b$  in B.
- $A \leq B$  is true if and only if the value comparison  $a \text{ lte } b$  is true for some item  $a$  in A and some  $b$  in B.
- $A > B$  is true if and only if the value comparison  $a \text{ gt } b$  is true for some item  $a$  in A and some  $b$  in B.
- $A \geq B$  is true if and only if the value comparison  $a \text{ gte } b$  is true for some item  $a$  in A and some  $b$  in B.

## Example

---

- For all books check if there is an author with surname "Stevens":

```
for $b in input()/bib/book
return $b/author/last = "Stevens"
```

Given the sample data this comparison is `true` for two instances of `book`, but `false` for the other ones.

## Related Syntax Construct

---

[ValueComp](#)

## GeneralStep

Create and possibly filter item sequence.

## Syntax

---



### GeneralStep

## Description

---

A `GeneralStep` can be part of a step expression. It consists of a primary expression that may be followed by a predicate expression (`StepQualifiers`). If a predicate expression is used, it selects one or more items.

## Examples

---

- The simplest form of a `GeneralStep` consisting only of a primary expression that turns out to be a numeric literal:

```
42
```

- Get a unique list of all doctors whose pager number start with "3":

```
let $a := input()/descendant::doctor[starts-with(@pager, "3")]
return distinct-values($a/name/surname)
```

In this example, `$a` is a `GeneralStep`.

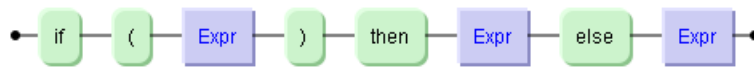
The other three occurrences are simple primary expressions: the first is a variable name ("`$a`") and the others are `QNames`.



## IfExpr

Conditional expression based on `if`, `then`, and `else`.

## Syntax



### IfExpr

## Description

An `IfExpr` implements a conditional expression. The expression following the `if` keyword is called the *test expression*, the expression following the `then` keyword is called the *then-expression* and the final one is called the *else-expression*. If the effective boolean value of the test expression is true, then the value of the then-expression is returned, otherwise the value of the else-expression is returned.

In contrast to other languages the else-expression is mandatory. If you do not need the else-expression, you can return an empty string, or a NaN value if you need a numerical value.

## Example

- Return all books with their titles and net prices:

```

for $a in input()/bib/book
let $vatrate := 16
let $net := if ($a/price) then xs:decimal($a/price) - xs:decimal($a/price) div ( ←
100 + $vatrate ) * $vatrate
                    else xs:double("NaN")
return
  <book>
    <title>{$a/title}</title>
    <price>
      <net>{ $net }</net>
    </price>
  </book>

```

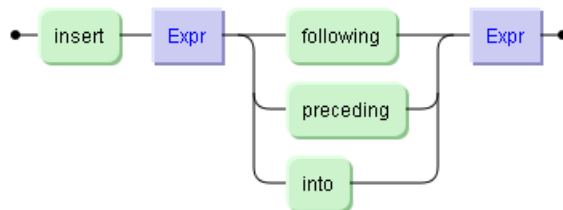
The conditional expression checks for the presence of a `book` child element `price` and returns the price minus the value added tax rate. Alternatively, you can place the `IfExpr` in the return clause:

```
for $a in input()/bib/book
let $vatrate := 16
return
  <book>
    <title>{$a/title}</title>
    <price>
      <net>{ if ($a/price) then xs:decimal($a/price) - xs:decimal($a/price) div ( ↵
100 + $vatrate ) * $vatrate
                                     else xs:double("NaN") }</net>
    </price>
  </book>
```

## InsertClause

Insert a node sequence at update position.

## Syntax



### InsertClause

## Description

An `InsertClause` is part of an `UpdateExpr` and contains two expressions: the first one evaluates to a node sequence to be inserted. The second one determines the update node at which the insert operation should take place. The effective result of the insert operation is one or more documents that contain the additional node sequence inserted at the respective update nodes. Depending on whether to include element or attribute nodes there are up to three ways of inserting a node sequence:

### following

At the update nodes insert the node sequence as following sibling in the document. This applies to inserting element nodes only.

### preceding

At the update nodes insert the node sequence as preceding sibling in the document. This applies to inserting element nodes only.

### into

At the update nodes insert the node sequence as child nodes in the document if element nodes are inserted. If the update node has not yet a child element node, the nodes then constitute the list of new element child nodes. If the update node already has child element nodes, they are inserted at the end of the current children nodes.

A sequence of attribute nodes is inserted as new attributes of the elements that are represented by the update nodes.

## Examples

---

- Add a new book as last child to bib:

```
update insert
<book year="2001">
  <title>XML Schema Part 0: Primer</title>
  <editor>
    <last>Fallside</last>
    <first>David C.</first>
    <affiliation>IBM</affiliation>
  </editor>
  <publisher>World Wide Web Consortium</publisher>
  <price/>
</book>
into input()/bib
```

- Add a new attribute edition into the book with the title "TCP/IP Illustrated".

```
update insert attribute edition {"1"}
into input()/bib/book[title = "TCP/IP Illustrated"]
```

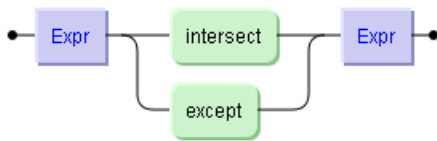
This update operation is not possible with the schema definition of the sample data as supplied. You need to enhance the schema accordingly before performing this update operation.

See also *Performing Update Operations in the XQuery4 User Guide*.

## IntersectExceptExpr

Combine node sequences.

### Syntax



**IntersectExceptExpr**

### Description

An `IntersectExceptExpr` combines two node sequences by returning a sequence in document order depending on the operator. Using the `intersect` operator all the nodes that occur in both sequences are returned in document order. Using the `except` operator all the nodes that occur in the first operand sequence, but not in the second.

### Examples

- The following query returns the empty sequence:

```
<A/> intersect <B/>
```

- The following query returns `<A/>`:

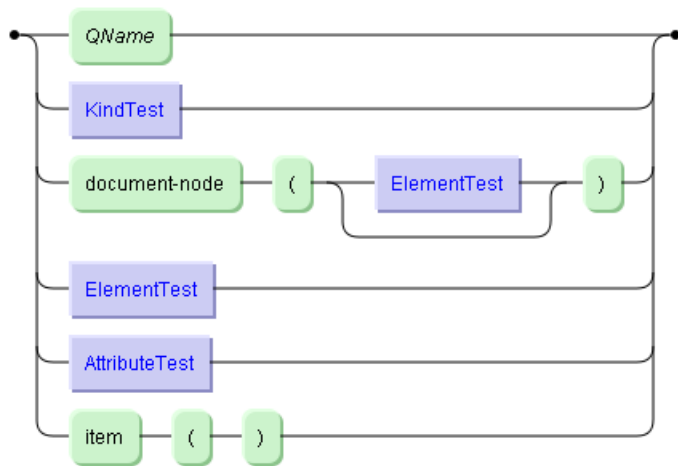
```
<A/> except <B/>
```

## ItemType

Specify the type of an item.

## Syntax

---



### ItemType

## Description

---

An `ItemType` specifies the type of an XQuery item by using one of:

- a qualified name that is interpreted as an atomic type,
- a `KindTest` matching processing instruction nodes, comment nodes, text nodes or any kind of nodes,
- any single item.

## Examples

---

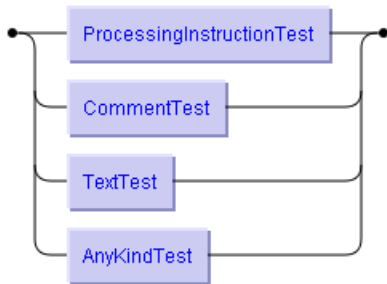
- The item type `xs:decimal` is a `QName` matching the value `47.11` since it is of the atomic type `xs:decimal`.
- `document-node(element(book))` matches a document node containing exactly one element node that is matched by the `ElementTest` `element(book)`.
- `item()` matches for example the atomic value `42` or the element `<bib>`.

## KindTest

Check the kind of node.

## Syntax

---



## KindTest

## Description

---

A `KindTest` is one of the two forms of a `NodeTest`. In a step expression it can test the selected nodes in the following ways:

- test whether the nodes are of a specific kind: `ProcessingInstructionTest` and `CommentTest`,
- select the text node children (`TextTest`),
- test for any kind of node (`AnyKindTest`),

## Example

---

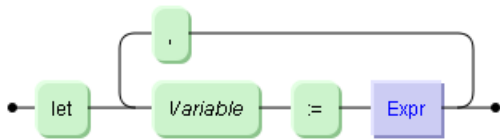
See the pages of the respective tests for examples.

## LetClause

Bind variables by evaluating expressions.

## Syntax

---



**LetClause**

## Description

---

A `LetClause` is part of `FLWOR` and `FLWU` expressions and contains one or more variables that will be related to the expression that follows. The `let` clause binds directly each variable to the result of evaluating the expression. If there are `for` clauses, the variable bindings are added to the tuples created in the `for` clauses. Otherwise a single tuple with all variable bindings is created. The `return` clause will be invoked once for each generated tuple that is retained after passing the optional `where` clause.

## Example

---

See [FLWORExpr](#) for examples.

## Related Syntax Constructs

---

[ForClause](#)   [WhereClause](#)



## LibraryModule

Provide XQuery fragment to other modules.

### Syntax

---



**LibraryModule**

### Description

---

A `LibraryModule` is one form of an XQuery module and contains XQuery code. It consists of a module declaration followed by a prolog which may be preceded by a pragma. In contrast to a main module a library module is not evaluated directly: it just provides the declarations of functions and variables which can later be imported into other modules. Especially it does not include a query body since that is reserved to the main module only.

### Related Syntax Construct

---

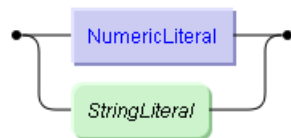
[ElementConstructor](#)

## Literal

Numeric or string literal.

## Syntax

---



Literal

## Description

---

In XQuery, a `Literal` is either a numeric literal or a string literal. See the section [Lexical Structure](#) for details.

## Example

---

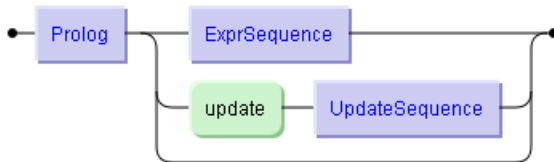
See the section [Lexical Structure](#) for examples.

## MainModule

Obligatory XQuery module with query body.

## Syntax

---



## MainModule

## Description

---

A query must always consist of exactly one `MainModule`. A main module contains a prolog and the actual query body. While a main module cannot include other main modules it can import library modules by specifying them in import declarations in the prolog.

## Related Syntax Construct

---

[LibraryModule](#)

## ModuleDecl

Identify library module.

### Syntax

---



#### ModuleDecl

### Description

---

A `ModuleDecl` identifies a module as a library module, since only library modules contain a module declaration. It contains the keyword `module` which is followed by a namespace declaration with the `NCName` specifying the namespace prefix and the string literal specifying the namespace URI that defines the target namespace of the library module. This namespace is used for all functions and variables exported by this module.

### Example

---

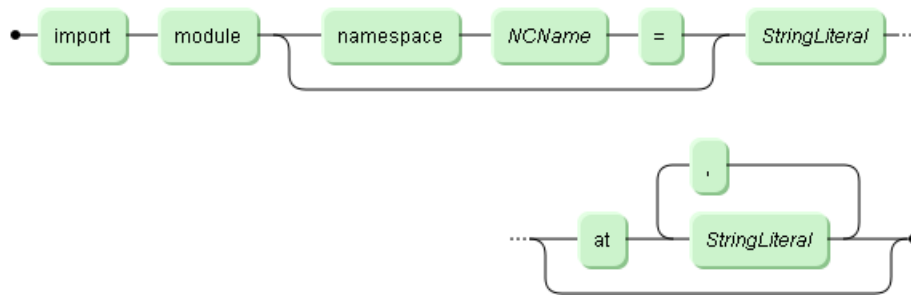
- This module declaration binds the URI *`http://example.org/math-functions`* to all functions and variable declarations that follow this declaration:

```
module namespace math = "http://example.org/math-functions"
```

## ModuleImport

Import one or more library modules.

## Syntax



## ModuleImport

## Description

A `ModuleImport` imports declarations of functions and variables from one or more library modules. The first string literal specifies the target namespace for which you can optionally define a namespace prefix. The `ModuleImport` imports all modules that share this target namespace. By using the `at` keyword you can provide an additional *location hint* with a string literal that must be a valid URL. The following rules apply for importing modules:

- It is an error if more than one module import in a prolog specifies the same target namespace.
- Module imports are not transitive which effectively means that you can access only the function and variable declarations directly contained in the imported module. So if module A imports module B and module B imports module C then module A cannot access the function and variable declarations of module C.
- The graph of module imports must not contain cycles. So if module A imports module B and module B imports module C then module C must not import module A.

## Example

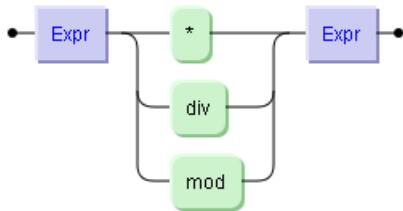
- This imports the module:

```
import module namespace math = "http://example.org/math-functions";
```

## MultiplicativeExpr

Multiply or divide numerical values.

### Syntax



### MultiplicativeExpr

### Description

The `MultiplicativeExpr` provides arithmetic operations for multiplying and dividing numerical values. Each operand is atomized so that it is either a single atomic value or an empty sequence. If either of the operands is atomized to an empty sequence, the result of the operation is also an empty sequence. If the two operands are of different types, type promotion is applied and the two operands are promoted to their least common type. The operation is then performed and either an atomic value is returned or a dynamic error such as division by zero issued.

The division operator is called `div`, since the typical division symbol `/` is already used in path expressions. The result of the `div` operation has the least common type of its operands unless both operands are integer values, in which case an atomic value of type `xs:double` is returned. The modulo operation (using the operator `mod`) returns the remainder of a division.

### Examples

- Compute the percentage of patients who have died:

```
count(input()//deceased) div count(input()/patient) * 100
```

Two multiplicative expressions are involved that are executed according to the normal precedence rule, since neither of them is enclosed in parentheses. The first operation divides the number of deceased patients by the total number of patients. The result of this operation is of type `xs:double`, since the atomized values of both operands are integer values. Consequently, the result of the following multiplication is also of type `xs:double`.

- Select all patients who were born in the first year of a decade:

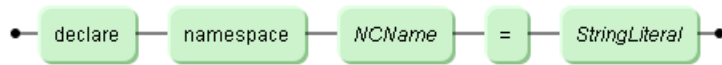
```
for $a in input()/patient
where $a/born mod 10 = 0
return $a
```



## NamespaceDecl

Declare a namespace.

### Syntax



#### NamespaceDecl

### Description

A `NamespaceDecl` is part of a prolog and declares a namespace for later use in the query body or module. It consists of the keywords `declare namespace`, an `NCName` that represents the namespace prefix, and a string literal, which is the namespace URI.

There is a number of namespace URIs predeclared that you can use without a namespace declaration of your own. These are:

Prefix	Namespace URI
<code>fn</code>	<code>http://www.w3.org/2002/08/xquery-functions</code>
<code>local</code>	<code>http://www.w3.org/2004/07/xquery-local-functions</code>
<code>tdf</code>	<code>http://namespaces.softwareag.com/tamino/TaminoDavFunction</code>
<code>tf</code>	<code>http://namespaces.softwareag.com/tamino/TaminoFunction</code>
<code>xdtd</code>	<code>http://www.w3.org/2004/07/xpath-datatypes</code>
<code>xf</code>	<code>http://www.w3.org/2002/08/xquery-functions</code>
<code>xml</code>	<code>http://www.w3.org/XML/1998/namespace</code>
<code>xs</code>	<code>http://www.w3.org/2001/XMLSchema</code>



**Note:** You can also declare a namespace directly in an element constructor, e.g.: `<library xmlns:dotcom="http://company.dot.com/namespaces/corporate"/>`

### Examples

- Declare namespace `http://www.w3.org/2001/XMLSchema` to use for constructor functions:

```
declare namespace xs = "http://www.w3.org/2001/XMLSchema"
```

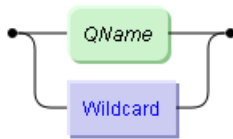
- Declare company-internal namespace:

```
declare namespace dotcom="http://company.dot.com/namespaces/corporate"
```

## NameTest

Select nodes by name or wildcard.

## Syntax



### NameTest

## Description

A `NameTest` is either a name, or a generic name that you can specify with a wildcard expression. It is used in a step expression to select a node sequence in the current step.

## Examples

- Select all books:

```
input()//book
```

The `book` element is specified by its `QName`.

- Retrieve Unicode transliterations for the current database:

```
declare namespace ino="http://namespaces.softwareag.com/tamino/response2"
input()//ino:transliteration/ino:*/@*
```

In this query there are three steps that involve a `NameTest`: The first one checks for the `QName` `transliteration` that is qualified by the `ino` namespace. The second name test uses the wildcard expression `ino:*` that is true for all element nodes that are in the `ino` namespace. The third name test uses the wildcard expression `@*` that is true for all attributes.

## Related Syntax Constructs

[StepExpr](#)

## NoAxisStep

Create and possibly filter node sequence on current axis.

### Syntax

---



### NoAxisStep

### Description

---

A `NoAxisStep` can be part of a relative path expression. It consists of a node test that may be followed by a predicate expression (`StepQualifiers`). Since you cannot specify an axis, it selects one or more nodes on the current axis.

### Related Syntax Constructs

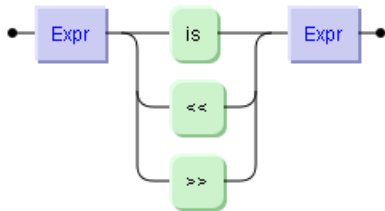
---

[AxisStep](#)   [GeneralStep](#)   [RelativePathExpr](#)   [StepExpr](#)   [StepQualifiers](#)

## NodeComp

Compare two nodes.

## Syntax



**NodeComp**

## Description

A `NodeComp` compares two nodes by their identity or by their document order. The following rules hold:

1. Each operand must be either a single node or an empty sequence.
2. If either operand is an empty sequence the result is an empty sequence.
3. A comparison using the `is` operator yields `true` if the two operand nodes have the same identity, otherwise `false` is returned.
4. A comparison using the `<<` operator yields `true` if the left operand node precedes the right operand node in document order, otherwise `false` is returned.
5. A comparison using the `>>` operator yields `true` if the left operand node follows the right operand node in document order, otherwise `false` is returned.

## Example

- The result of this comparison is `false`, because each node has its own identity (rule 3):

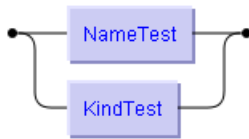
```
<a>42</a> is </a>
```

## NodeTest

Check if nodes satisfy conditions on the kind or name.

## Syntax

---



### NodeTest

## Description

---

A `NodeTest` is used in a step expression and determines kind or name of one or more nodes. You can either specify the name or the type of nodes.

If you specify the name, then nodes of the principal node kind for the current axis will be selected for this step. For the attribute axis, attribute nodes will be selected, for all other axes element nodes will be selected.

If you specify `node()` using `KindTest`, then all nodes on the current axis will be selected. Otherwise only the nodes of that kind, if present on the current axis, will be selected.

## Example

---

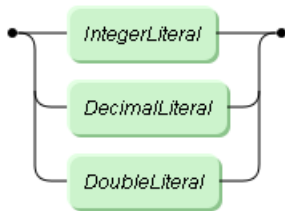
See [KindTest](#) and [NameTest](#) for examples.

## NumericLiteral

Numeric literal.

### Syntax

---



**NumericLiteral**

### Description

---

A `NumericLiteral` is a literal that denotes an integer value (`IntegerLiteral`), a decimal value (`DecimalLiteral`) or a double precision value (`DoubleLiteral`).

### Example

---

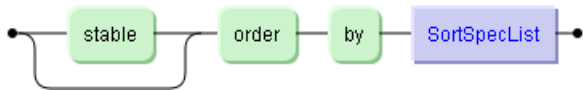
See the section [Lexical Structure](#) for examples and the lexical (syntactic) definition.



## OrderByClause

Order intermediate results.

### Syntax



**OrderByClause**

### Description

In a FLWOR expression, an `OrderByClause` sorts the intermediate result tuples before passing them to the `return` clause where the output sequence is constructed.

### Example

See [FLWORExpr](#) and [SortSpecList](#) for examples.

- List book titles with their year of publication, newest first:

```
for $a in input()/bib/book
order by $a/@year descending
return ($a/@year, $a/title)
```

- List book titles sorted according to French collation:

```
for $a in input()/bib/book
order by $a/title ascending collation "collation?language=fr"
return $a/title
```

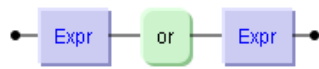
Since ascending is the default order modifier, the `order by` clause could also read:

```
order by $a/title collation "collation?language=fr"
```

## OrExpr

Check if one or both of two expression are logically true.

## Syntax



### OrExpr

## Description

An `OrExpr` is a logical expression that is evaluated by determining the effective boolean value (ebv) of each of its operands. The value of the `OrExpr` is then determined as follows:

or expression	ebv(op2) = true	ebv(op2) = false	error in ebv(op2)
ebv(op1) = true	true	true	true or error
ebv(op1) = false	true	false	error
error in ebv(op1)	true or error	error	error

## Example

- Select all patients born in 1950 or 1960:

```

for $a in input()/patient
where $a/born = 1950 or $a/born = 1960
return $a

```

## Related Syntax Construct

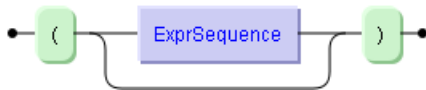
[AndExpr](#)

## ParenthesizedExpr

An expression sequence enclosed in parentheses.

### Syntax

---



**ParenthesizedExpr**

### Description

---

You can use a `ParenthesizedExpr` to control the order of evaluation in an expression with multiple operators.

A `ParenthesizedExpr` can also server as a sequence constructor, i.e. the parentheses can contain zero or more expressions that comprise a sequence.

### Example

---

- The following two queries have different results:

```
(count(input()/bib/book) +  
  count(input()/reviews/entry)) * 2  
  
count(input()/bib/book) + count(input()/reviews/entry) * 2
```

In the first query the addition operator is evaluated before the multiplication operator because of the parentheses. In the second query no parentheses are used so the multiplication operator is evaluated first.

## PathExpr

Select nodes on a path.

## Syntax

---



**PathExpr**

## Description

---

A `PathExpr` creates a sequence of nodes by following a path from a starting point that is specified either by:

- an absolute path expression, which has `/` as its first character and starts at the context node's root node, or by:
- a relative path expression, which starts at the context node.

## Example

---

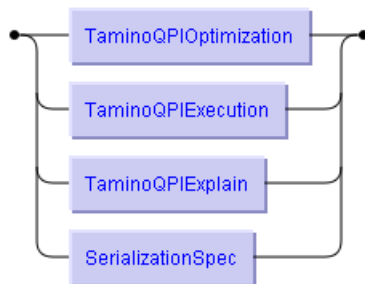
See [AbsolutePathExpr](#) and [RelativePathExpr](#) for examples.

## Pragma

Query Processor Directive.

## Syntax

---



### Pragma

## Description

---

A Pragma is part of the query prolog and provides directives for processing the query. In general, pragmas are *extension expressions* which are expressions whose semantics are implementation-defined. In Tamino, pragmas are syntactically similar to processing instructions. However, they are called *query processing instructions* (QPI) and are denoted by curly braces as delimiters instead of angle brackets:

```
"{?" + PragmaName + " " + PragmaContents + " "*" + "?"}"
```

Note that the pragma name must immediately follow the opening delimiter while this is not required for the closing delimiter, so `{?explain?}` is valid, while `{? explain ?}` is not. The following pragmas are available

- `TaminoQPIOptimization` sets parameter for optimizing certain parts of query processing.
- `TaminoQPIExecution` sets parameter for tuning execution of user-defined functions.
- `TaminoQPIInline` determines a strategy for inlining user-defined functions.
- `TaminoQPIExplain` retrieves the query execution plan.
- `SerializationSpec` determines the way Tamino serializes the query result.

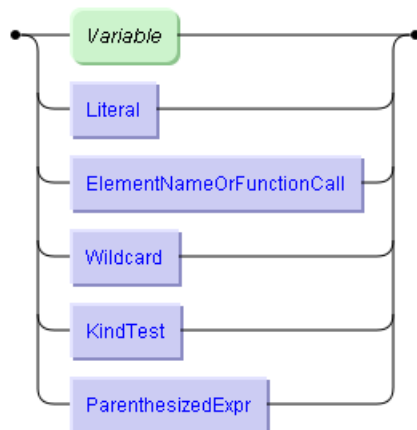


**Note:** The syntax proposed for pragmas in recent working drafts of the XQuery standard is not supported so that using `("::"` and `::")` as delimiters yields a syntax error.

## PrimaryExpr

Basic XQuery primitive.

## Syntax



### PrimaryExpr

## Description

A `PrimaryExpr` is composed of the language primitives of XQuery. These include variables, literals, function calls, and parenthesized expressions.

Variables consist of a "\$" prefix character and a variable name, which must be a `QName`. Literals are divided into string literals and some forms of numeric literals. An `ElementNameOrFunctionCall` either represents a simple element name, which is a `QName` or a call of a function defined in *Tamino XQuery 4*. Wildcard expressions and `KindTest` are used in the context of path expressions, while an expression that is enclosed in parentheses influences the precedence of the operators.

## Examples

- Some literal expressions

```
42  
42.0  
"42"  
42E2
```

The first literal denotes the integer value 42, the second literal denotes the decimal value 42.0. The third literal is a string literal that is composed of the characters "4" and "2", while the last literal denotes a value of double precision.

- Variables are used in FLWOR and FLWU expressions such as the following:

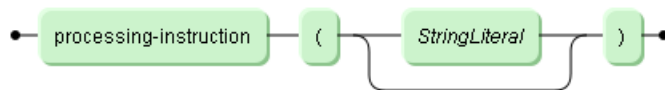
```
for $a in input()/bib/book  
where $a/@year >= 2000  
return $a/title
```



## ProcessingInstructionTest

Check for processing instruction nodes.

### Syntax



### ProcessingInstructionTest

### Description

A `ProcessingInstructionTest` is one of a number of node tests that are used in a step expression. The expression `processing-instruction()` is true for any processing instructions. You can also specify a string literal that must be equal to the value of the target application for the test to succeed.

### Examples

- Select all processing instructions in the current collection:

```
input()//processing-instruction()
```

↩

- Select all processing instructions that use the target application "xm-replace\_text":

```
input()//processing-instruction("xm-replace_text")
```

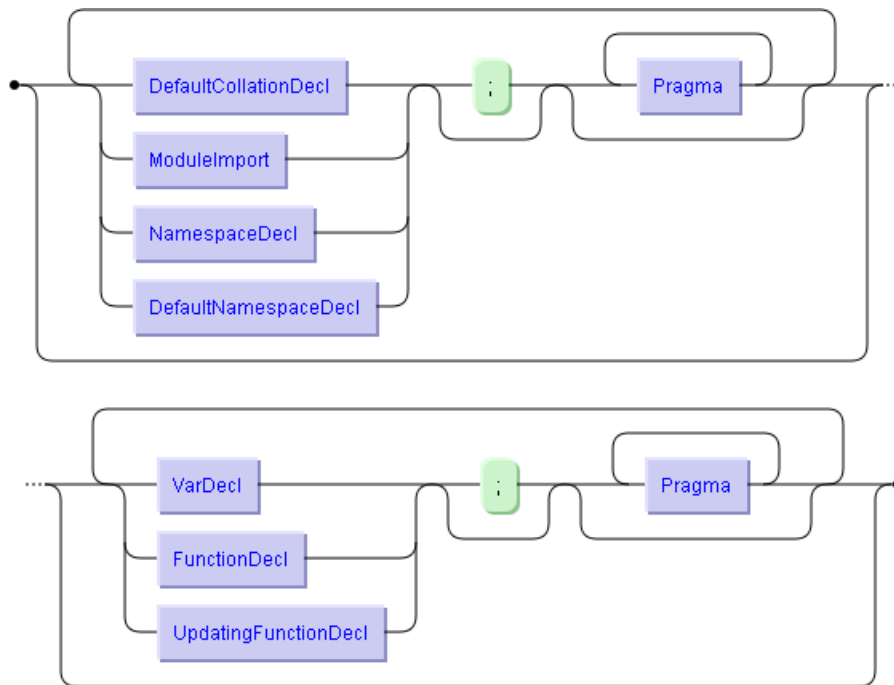
↩

## Prolog

Create environment for query processing.

## Syntax

---



### Prolog

## Description

---

The `Prolog` prepares the environment in which the query processing takes place. The environment is defined for the module that contains the prolog, since the main module as well as any library module may contain their own prolog. A prolog is roughly organized into two parts. The first part contains module imports and several declarations that affect either query processing (`DefaultCollationDeclaration`) or the interpretation of `QNames` within the query (`NamespaceDeclaration` and `DefaultNamespaceDeclaration`). The second part consists of declarations of variables and functions. They appear at the end of the prolog because the imports and declarations in the first part may have impact on them.

Declarations and imports may all be followed by one or more pragmas each of which is followed by a semicolon and optional pragmas. These are directives for the query processor and control the way Tamino processes the query and outputs the query result.

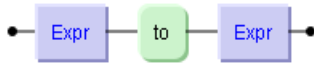
Declarations, module imports and pragmas can only occur in the query prolog. It is an error to place them in the query body.

## RangeExpr

Construct a sequence of consecutive integers.

### Syntax

---



#### RangeExpr

### Description

---

A `RangeExpr` is used to construct a sequence of consecutive integers. Each of the two operand expressions is expected to be of type `xs:integer`. If either operand is an empty sequence, or if the first integer operand is larger than the second, the result of the range expression is an empty sequence. Otherwise, the result is a sequence beginning with the first integer operand and all consecutive integers including the second integer operand.

### Examples

---

- This range expression is used as one operand in constructing a sequence. The result is the sequence (42, 8, 9, 10, 11).

```
(42, 8 to 11)
```

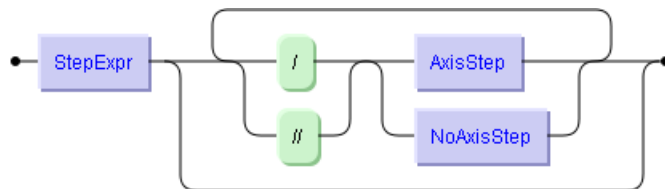
- This range expression is evaluated to a sequence with the single integer value 7:

```
7 to 7
```

## RelativePathExpr

Select nodes on the relative path.

### Syntax



**RelativePathExpr**

### Description

A `RelativePathExpr` is a path expression consisting of a step expression that can be followed by one or more steps with or without axis. Starting from the context node, `/` separates location steps, while `//` is the abbreviated syntax for `/descendant-or-self::node()`. If followed by an `AxisStep`, you can specify an abbreviated or unabbreviated step. Otherwise, you can specify a node test plus optional predicate expression.

### Examples

- Select all books written by Stevens:

```
input()//book[author/last = "Stevens"]
```

The relative path in the bracketed filter expression starts at the context node `book` and selects the `author` child node (a `StepExpr`), and then its child node `last` (a `NoAxisStep`).

- Select all patients who have been treated with therapies including medication in the form of tablets:

```
input()/patient[therapy//@form = "tablet"]
```

The relative path in this expression starts at with `therapy` child nodes of `patient` elements and selects all descendant attribute nodes with the name `form` and value `"tablet"` (an abbreviated `AxisStep`).

## RenameClause

Rename the nodes of a node sequence.

### Syntax



### RenameClause

### Description

A `RenameClause` is part of an `UpdateExpr` and renames the nodes of the node sequence that is the result of the evaluation of the expression with the supplied `QName`. A rename operation can be applied to element, attribute and processing instruction nodes.

### Example

- Rename the attribute `year` as `jahr`:

```
update rename input()/bib/book/@year as jahr
```

This rename operation will fail for the provided sample data and schema definitions, since an attribute `jahr` is not included in the schema. See *Performing Update Operations in the XQuery4 User Guide* for more information.

## ReplaceClause

Replace a node.

## Syntax

---



### ReplaceClause

## Description

---

A `ReplaceClause` is part of an `UpdateExpr` and contains two expressions: the first one determines the update node at which the replace operation should take place. The second expression evaluates to a node that is used as replacement.

## Examples

---

- Replace the `title` of the book "TCP/IP Illustrated":

```
update replace input()/bib/book/title[. = "TCP/IP Illustrated" ]
  with <title>TCP/IP Illustrated I. The Protocols</title>
```

Any `title` element nodes that satisfy the predicate expression will be replaced by the new `title` node.

- Replace the `year` attribute of the book with the title "TCP/IP Illustrated":

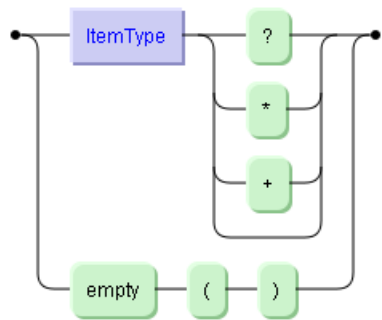
```
update replace input()/bib/book[title = "TCP/IP Illustrated"]/@year
  with attribute year {"2003"}
```



## SequenceType

Specify the type of an XQuery value.

## Syntax



## SequenceType

## Description

A `SequenceType` specifies the type of an XQuery value by identifying it either as an item type together with an occurrence indicator or as an empty sequence using `empty()`. An item type can be the qualified name or any of the node tests available in Tamino XQuery. Since a value is always a sequence, you can use an occurrence indicator specifying the number of items in the sequence:

- `?` specifies one or zero items
- `*` specifies zero or more items
- `+` specifies one or more items

## Examples

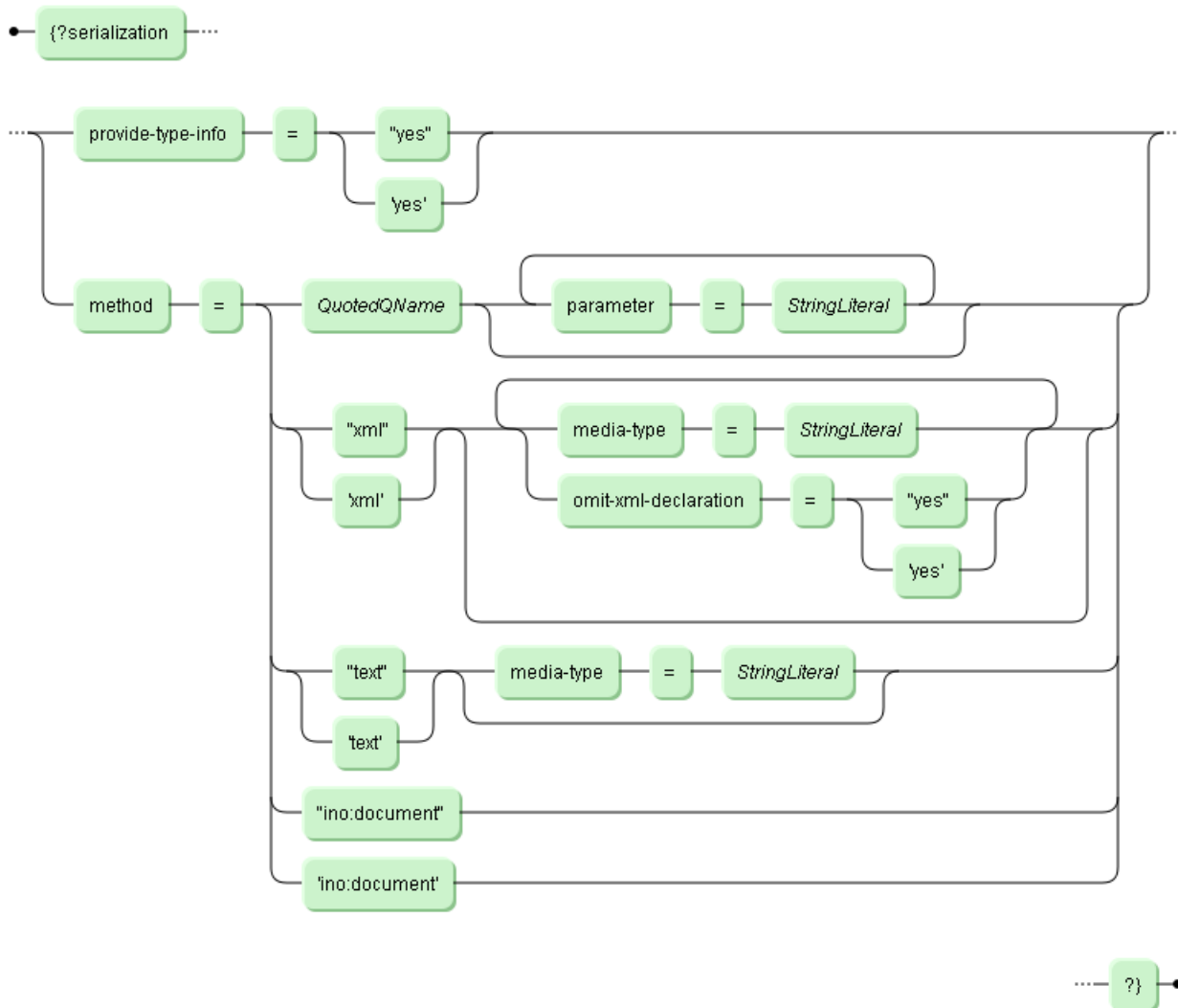
Some sequence types that you can use in XQuery expressions:

- `xs:date` refers to the built-in atomic schema type named `xs:date`
- `attribute()?` refers to an optional attribute
- `element(name)` refers to an element called `name` of any type
- `node()*` refers to a sequence of zero or more nodes of any type
- `item()+` refers to a sequence of one or more nodes or atomic values

## SerializationSpec

Specify serialization of query output.

### Syntax



### SerializationSpec

### Description

A `SerializationSpec` is a query pragma that has the form of a QPI (query processing instruction), i.e., it is enclosed by `"{"` and `"}"`. It specifies how Tamino returns the output of a query by applying

a serialization method. One common effect is that Tamino does not wrap the output with the standard response wrapper using the document element `ino:response`.

The following methods for serializing the query output are available:

#### *QuotedQName*

Specify an output handler that performs serialization in place of the standard XQuery processor. It must have been registered as a server extension and can take zero or more parameters separated by whitespace. Since the name of the output handler is the value of `method`, it cannot be "text" or "xml" (names are case-sensitive).

#### `text`

The query output is treated as text. If it is not followed by a media type then "text/xml" is assumed and set as the Content-Type value in the HTTP response header.

#### `xml`

The query output is treated as XML. If it is not followed by a media type then "text/xml" is assumed and set as the Content-Type value in the HTTP response header. Although the query output is expected to be XML there are two exceptions that may violate the rules of an XML document:

- **No XML Declaration**

You can use the additional option `omit-xml-declaration="yes"` to omit the XML declaration. Note that no value other than "yes" is allowed, but you may use single quote characters instead of double quote characters. This is for example useful when embedding the query result as a tree fragment into other XML documents.

- **Multiple Document Nodes**

The XML specification requires a single document element, but you may output more than one document element. Normally this is prevented by the Tamino response wrapper.

#### `ino:document`

retrieve non-XML documents

## Examples

- Report patient records in a way that is determined by an XSL stylesheet:

```
{?serialization method="XSLT.transform"
  parameter="stylesheets" parameter="stylesheets/patientRecords.xsl"?}
<report>
  { for    $a in collection('Hospital')/patient
    return <patient>{ $a/name, $a/sex, $a/born, $a/address }</patient>
  }
</report>
```

- Consider the following query:

```
let    $a := <A attr=''>&lt;</A>
let    $b := <B/>
return ($a,$b)
```

Tamino returns the following document: (It is indented for better readability; please note that the encoding value may differ depending on your browser settings):

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<ino:response xmlns:ino="http://namespaces.softwareag.com/tamino/response2"
  xmlns:xql="http://metalab.unc.edu/xql/"
  <xq:query xmlns:xq="http://namespaces.softwareag.com/tamino/XQuery/result">
    <![CDATA[let    $a := <A attr=''>&lt;</A>
                    let    $b := <B/>
                    return ($a,$b)]]></xq:query>
  <ino:message ino:returnValue="0">
    <ino:messageline>XQuery Request processing</ino:messageline>
  </ino:message>
  <xq:result xmlns:xq="http://namespaces.softwareag.com/tamino/XQuery/result">
    <A attr="&quot;">&lt;</A>
    <B/>
  </xq:result>
  <ino:message ino:returnValue="0">
    <ino:messageline>XQuery Request processed</ino:messageline>
  </ino:message>
</ino:response>
```

As an effect of the Tamino response wrapper the resulting nodes `<A>` and `<B>` are child element nodes of `xq:result`. Also, named entity references are used for the five character references predefined in XML. Now we repeat the query and specify serialization methods. First, we use "xml":

```
{?serialization method="xml"?}
let    $a := <A attr=''>&lt;</A>
let    $b := <B/>
return ($a,$b)
```

The result is an XML document that is no longer well-formed, since it contains two document elements.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<A attr="&quot;">&lt;</A><B></B>
```

Using the serialization method "text" yields a different output: The XML declaration is not used and the XML entity references are resolved now:

&lt;

■ Retrieve basic book information in tabular form:

```
{?serialization method="text" media-type="text/plain"?}
declare namespace tf = "http://namespaces.softwareag.com/tamino/TaminoFunction"
declare namespace xs = "http://www.w3.org/2001/XMLSchema"
for $book in input()/bib/book
let $year      := <year>{ tf:createTextNode(xs:string($book/@year)) }</year>
let $authors :=
  for $i in $book/author
  return ( $i/last/text(), " ", $i/first/text(), "&#x09;" )
return
( "Published: ", $year/text(), "&#x0A;",
  "Author(s): ", $authors, "&#x0A;",
  "Title      : ", $book/title/text(), "&#x0A;&#x0A;" )
```

This query is in some respect similar to the one used to exemplify [EnclosedExpr](#), but delivers a pure text instead of an XML document. Here, multiple author names are separated by a horizontal tab character (U+0009):

```
Published: 1994
Author(s): Stevens, W.
Title      : TCP/IP Illustrated

Published: 1992
Author(s): Stevens, W.
Title      : Advanced Programming in the Unix environment

Published: 2000
Author(s): Abiteboul, Serge Buneman, Peter Suciu, Dan
Title      : Data on the Web

Published: 1999
Author(s):
Title      : The Economics of Technology and Content for Digital TV
```

The difficult part is extracting the year attribute. A straightforward version of this query would be to directly return the attribute:

```
{?serialization method="text" media-type="text/plain"?}
for $book in input()/bib/book
let $authors :=
  for $i in $book/author
  return ( $i/last/text(), " ", $i/first/text(), "&#x09;" )
return
( "Published: ", $book/@year, "&#x0A;",
  "Author(s): ", $authors, "&#x0A;",
  "Title      : ", $book/title/text(), "&#x0A;&#x0A;" )
```

But this yields an error, since an attribute can not be serialized without an element context and here a sequence is returned. Therefore, the variable `$year` is used: it contains an element `year` whose content is the attribute value as a string. The text node of the `year` element is then used in the `return` clause.

## Related Syntax Construct

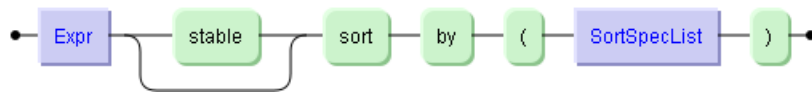
---

[Prolog](#)

## SortExpr

Sort a sequence of items.

## Syntax



SortExpr

## Description

A `SortExpr` sorts a sequence of items that is specified as the expression to the left of the keywords `sort` `by` according to an *ordering expression*, the `SortSpecList`. The result of evaluating the expression that precedes the keywords `sort` `by` is the *input sequence*. The sort result is the *output sequence*.

The items of the input sequence are reordered according to the ordering expressions in the `SortSpecList`. If there is more than one ordering expression, then the first one is the primary sort criterion and the other expressions are then processed from left to right. For each ordering expression you can specify a sort direction, either *ascending* or *descending*, with *ascending* as default.

For the sort operation to succeed, a common supertype of each ordering expression is determined which is used for sorting. Each expression must return not more than one item per object to be sorted. Also the comparison operator `gt` must be defined for this type, otherwise an error is raised.

Using the keyword `stable` does not have any effect and so the keyword can be left out.

## Example

- Sort all books that cost less than 50 dollars by first author and title:

```

declare namespace xs="http://www.w3.org/2001/XMLSchema"
for $a in input()/bib/book
where xs:decimal($a/price) < 50
return $a
sort by (author[1]/last, title)

```

The input sequence comprises all `book` elements that cost less than 50 dollars. The primary sort criterion is `author[1]/last`, so the input sequence is ordered by the `last` child element of the

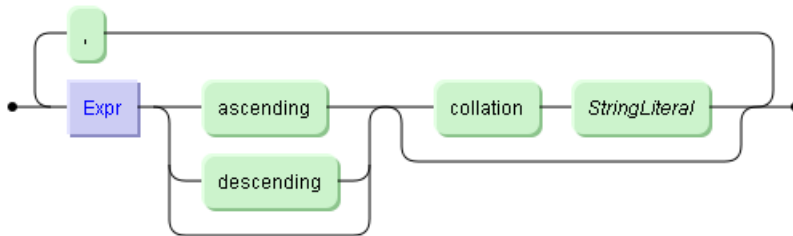
first author element. Within each author group, the input sequence is ordered by the title element.



## SortSpecList

Define ordering expression and sort direction.

### Syntax



### SortSpecList

### Description

A `SortSpecList` defines one or more sort criteria along with the direction of ordering, separated by commas. Each `SortSpecList` consists of an ordering expression that is optionally followed by one of the keywords `ascending` or `descending`. If you omit the ordering direction, `ascending` is taken as default. You can provide optional collation information introduced by the keyword `collation` and an appropriate string literal. The first `SortSpecList` acts as the primary sort criterion, further `SortSpecLists` as secondary criterion etc.

### Example

See [SortExpr](#) for examples.

- List book titles with their year of publication, newest first:

```
for $a in input()/bib/book
order by $a/@year descending
return ($a/@year, $a/title)
```

- List book titles sorted according to French collation:

```
for $a in input()/bib/book
order by $a/title ascending collation "collation?language=fr"
return $a/title
```

Since `ascending` is the default order modifier, the `order by` clause could also read:

```
order by $a/title collation "collation?language=fr"
```

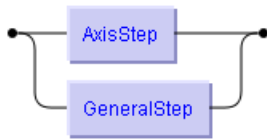
See also [SortExpr](#) for examples.

## StepExpr

Create and possibly filter a sequence of items.

## Syntax

---



**StepExpr**

## Description

---

A `StepExpr` creates a sequence of items. It may consist of a general step that includes primary expressions or of a step on some axis in forward or reverse direction. This sequence can be filtered by using predicate expressions that are defined in [StepQualifiers](#). The value of the step is determined by those sequence items that meet the predicate condition.

## Example

---

See [AxisStep](#) and [GeneralStep](#) for examples.

## Related Syntax Constructs

---

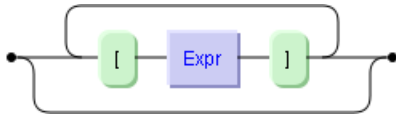
[PrimaryExpr](#)

## StepQualifiers

Restrict item sequence to those matching one or more predicates.

### Syntax

---



#### StepQualifiers

### Description

---

In a step expression, `StepQualifiers` can be used to restrict the current sequence of items with the help of one or more predicate expressions each of which is enclosed in brackets. If you specify more than one step qualifier, their order has no effect on the result except if you use numeric predicates.

The predicate expression can be either a Boolean or a numeric expression. A numeric predicate `[n]` is short for `[position()=n]` which is true for all nodes that are the *n*th child element in the input node sequence. If the type of the predicate expression is not numeric, then it is treated as a Boolean expression.

### Examples

---

- Select all books published by one or more editors before 2000:

```
input()//book[editor][@year < 2000]
```

The sequence of all `book` elements in the current collection is filtered by two consecutive step qualifiers: the first predicate expression is true for all `book` elements that have a child element `editor`, which means that all `book` elements with a child element `author` are filtered out. The retained node sequence is then checked for the second predicate. Note that the context node is still `book` and not `editor` so the path expression looks for an attribute `year` of the `book` element to see if its value is less than 2000.

- Select the third book:

```
input()//book[3]
```

This query is short for `input()//book[position() = 3]`. From all book elements the one that is the third child element of its parent is retained.

## Related Syntax Construct

---

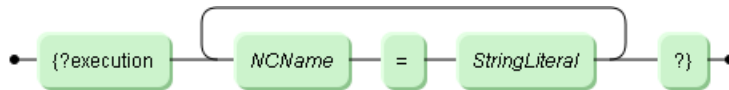
[StepExpr](#)

## TaminoQPIExecution

Determine parameters of query execution.

### Syntax

---



#### TaminoQPIExecution

### Description

---

A `TaminoQPIExecution` is a query pragma that has the form of a QPI (query processing instruction), i.e., it is enclosed by "{?" and "?}". You can use it to take influence on the query execution itself. This is useful to avoid running out of memory during the execution of user-defined functions, especially if they use recursion. The following parameters are supported:

<code>call-stack-depth</code>	restrict the depth of the stack for user-defined functions (default value is 256)
<code>document-cache-size</code>	maximum cache size during query execution in MB; this way you can use it flexibly for single queries instead of the global server parameter <code>XQuery_document_cache_size</code>
<code>memory</code>	amount of memory available to a single query processing thread during query execution in MB; it cannot be less than the maximum cache size determined by the value of either the execution parameter <code>document-cache-size</code> or of the server parameter <code>XQuery_document_cache_size</code> .

### Example

---

- The following execution pragma sets the available memory per query thread to 100MB and the maximum call stack depth to 1000. The document cache size is not set here and therefore determined by the value of the server parameter `XQuery_document_cache_size`:

```
{?execution memory="100" call-stack-depth="1000?"}
```

## TaminoQPIExplain

Retrieve query execution plan.

### Syntax

---



#### TaminoQPIExplain

### Description

---

A `TaminoQPIExplain` is a query pragma that has the form of a QPI (query processing instruction), i.e., it is enclosed by "{?" and "?}". If you use this pragma, the query will not be executed. Instead, an XML representation of the execution plan for the query expression is returned in the result document. You can use this information to determine which access paths will be used for Tamino data during query execution.

The following operator elements are of interest:

<code>XqcUnnestCollectionScan</code>	retrieve all documents of the current collection
<code>XqcUnnestDoctypeScan</code>	retrieve all documents of a doctype
<code>XqcUnnestStdIdxScan</code>	index-based retrieval of documents

The doctype scan operator contains information on which doctype it refers to. The index scan operator additionally contains information on the actual index, the index comparison operator and the comparison value.

### Example

---

```
{?explain?} input()/entry[@name="Hoch Franz"]
```

The result document contains this `XqcUnnestStdIdxScan` operator:



```
<XqcUnnestStdIdxScan outputvariable="# 1,1">  
- <XqcAlgProperties>  
  <XqcResultSchema variables="# 1,1" />  
  <XqcFreeVarSet variables="" />  
  <XqcCollScanSpec collections="" />  
</XqcAlgProperties>  
- <XqcAlgStdIdxPredicate lhsVar="($2,10)" doctype="entry" node="name" compareOp="=">  
  <XqcAlgLiteralFunction value="Hoch Franz" />  
</XqcAlgStdIdxPredicate>  
</XqcUnnestStdIdxScan>
```

## TaminoQPIInline

Use default inlining strategy for user-defined functions.

### Syntax

---



#### TaminoQPIInline

### Description

---

A `TaminoQPIInline` is a query pragma that has the form of a QPI (query processing instruction), i.e., it is enclosed by "{?" and "?}". It is a hint to the query processor to use the default inlining strategy for processing the immediately following user-defined function. This means that a function call is replaced by the code of the called function.

If inlining is not possible, the inline pragma is ignored.

See also the *Performance Guide* for more information about optimizing processing of user-defined functions.

### Example

---

- This recursive function should be processed using the default inlining strategy:

```
module namespace math = "http://example.org/math-functions"
{?inline?}
declare function math:power($b as xs:integer, $e as xs:integer) as xs:integer
{
  if ($e <= 0)
  then 1
  else math:power($b, $e - 1) * $b
}
```

### Related Syntax Construct

---

[TaminoQPIOptimization](#)

## TaminoQPIOptimization

Choose query optimization strategy.

### Syntax



**TaminoQPIOptimization**

### Description

A `TaminoQPIOptimization` is a query pragma that has the form of a QPI (query processing instruction), i.e., it is enclosed by "{" and "}". It directs the query processor to use an optimization strategy.

As optimization strategy you can use inlining of user-defined functions to minimize the overhead of function calls and to enable optimization across function boundaries. The parameter is `inline` and the following values are allowed:

- "none" no inlining of user-defined functions is performed
- "default" inlining of user-defined functions is performed when the `inline` hint is used
- "full" all user-defined functions are inlined except those that directly or indirectly reference themselves

### Example

- The following query directs the query processor to use full inlining to a query:

```
{?optimization inline="full"?}
import module namespace math="http://example.org/math-functions"
math:power(2, 2)
```

### Related Syntax Construct

[TaminoQPIInline](#)

## TextTest

Check for text nodes.

## Syntax

---



### TextTest

## Description

---

A `TextTest` restricts the node sequence to text nodes. This test is not meaningful on the attribute axis, since attributes have no text nodes.

## Example

---

- Select the text nodes of all patient's surnames:

```
input()/patient/name/surname/text()
```

Text nodes appear as `xq:textNode` in the response document returned by Tamino:

```
- <xq:result xmlns:xq="http://namespaces.softwareag.com/tamino/XQuery/result">
  <xq:textNode>Atkins</xq:textNode>
  <xq:textNode>Bloggs</xq:textNode>
  <xq:textNode />
</xq:result>
```

## TreatExpr

Modify static type of operand.

## Syntax



**TreatExpr**

## Description

A `TreatExpr` modifies the static type of its operand and takes an expression and a sequence type as arguments. It does not change the dynamic type; rather, it ensures that an expression has an expected dynamic type at evaluation time.

## Example

- The static type of `$myaddress` may be `Address`, a less specific type than `USAddress`. However, at run-time, the value of `$myaddress` must match the type `USAddress` using rules for Sequence-Type matching; otherwise a dynamic error is raised.

```
$myaddress treat as element()
```

## UnaryExpr

Negate numerical value of an expression.

## Syntax

---



**UnaryExpr**

## Description

---

A `UnaryExpr` consists of the unary operator and an expression. The expression is atomized so that it is either a single atomic value or an empty sequence. If it is an empty sequence, the result of the operation is also an empty sequence.

The unary operator has higher precedence than binary operators.

## Examples

---

- An atomic integer value:

```
- 3
```

This is different from `-3`, since that is interpreted as a single atomic value, while `- 3` is parsed as a unary expression. Effectively, both expressions yield the negative integer value `-3`.

- Change the sign of the numeric value `-3`:

```
- ( - 3 )
```

- Subtract the year of birth of each patient from 2004 and change it to a negative value:

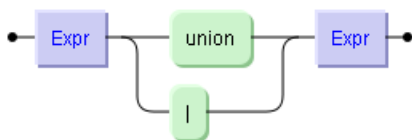
```
for $a in input()/patient/born  
return -(2004 - $a)
```

## UnionExpr

Combine node sequences.

### Syntax

---



### UnionExpr

### Description

---

A `UnionExpr` combines two node sequences by returning a sequence in document order with all the nodes that occur in either of both sequences. Duplicate nodes from the result sequences based on node identity are eliminated.

You can use one of the operands `union` and `|`: the effect is the same.

### Examples

---

- The following query returns the sequence (`<A/>`, `<B/>`)

```
<A/> union <B/>
```

- The following query returns the sequence (`<B/>`, `<B/>`), since the element constructors create two element nodes with the same name, but different identity:

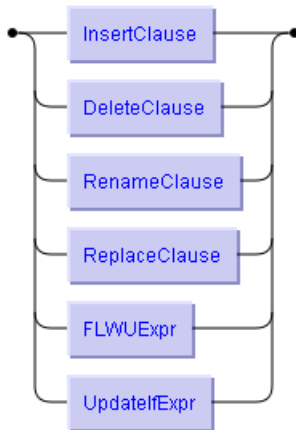
```
<B/> union <B/>
```



## UpdateExpr

Update expression.

### Syntax



### UpdateExpr

### Description

An `UpdateExpr` contains either one of the basic update operations that insert, delete, rename or replace node sequences. Or it is a `FLWUExpr` which is a variant of the regular FLWR expressions for more complex update operations. All update expressions are introduced by the keyword `update`.

As a result the list of those XML objects is returned that were affected by the update operation.

For any update operation, you need appropriate writing permission for the resulting document. The resulting document must still conform to the schema definition except if you deliberately turn off schema validation for the affected database. If you use more than one update operation in a `FLWUExpr` this can lead to conflicts. See [Performing Update Operations](#) for details.

### Example

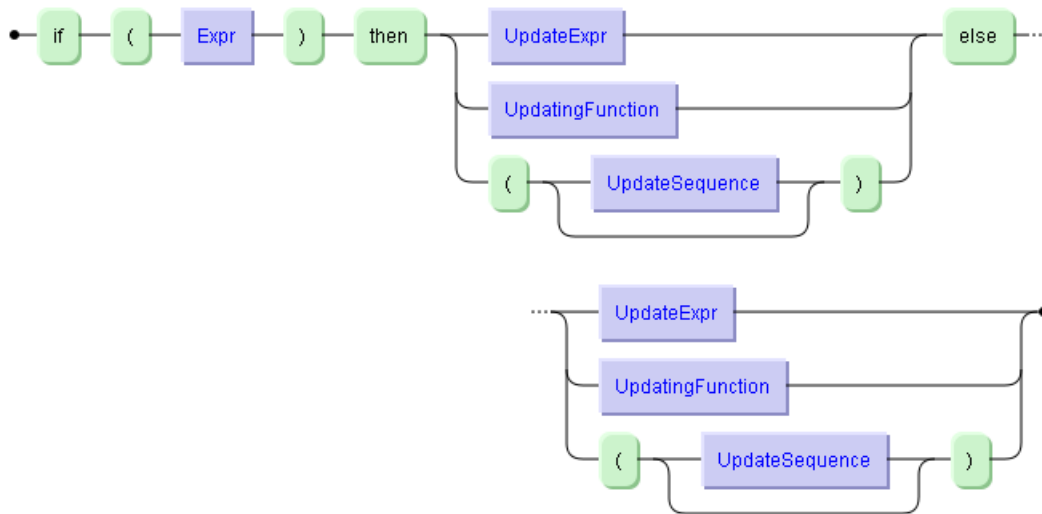
See the pages to the respective update expressions for examples.

## UpdateIfExpr

An `UpdateIfExpr` is a conditional expression for use in an update expression. If the expression following the `if` keyword evaluates to true, the update expression following the `then` keyword is applied, otherwise the update expression following the `else` keyword is applied.

## Syntax

---



## Description

---

If the `then` branch and/or the `else` branch does not require any update operations, an empty pair of parentheses should be used.

Note that the `else` clause is mandatory.

## Example

---

Insert an attribute, if it does not exist yet, otherwise replace the existing attribute, if its value is less than 100. This query contains two `UpdateIfExpr` constructs, where the second one occurs in the `else` branch of the first:

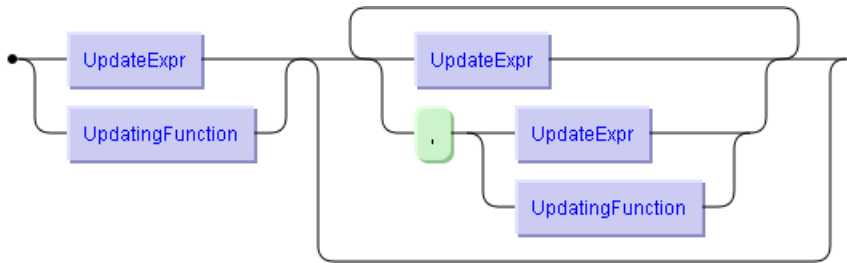
```
update
  for $book in input()/bib/book
  do
    if (empty($book/@year)) then
      insert attribute year {year-from-date(current-date())}
      into $book
    else if ($book/@year < 100) then
      replace $book/@year
      with attribute year {1900 + $book/@year}
    else
      ()
```

## UpdateSequence

Sequence of update expressions.

### Syntax

---



**UpdateSequence**

### Description

---

An `UpdateSequence` consists of one or more update expressions that you can use to insert, delete, rename or replace node sequences.

### Example

---

See [UpdateExpr](#) for information about examples.

### Related Syntax Constructs

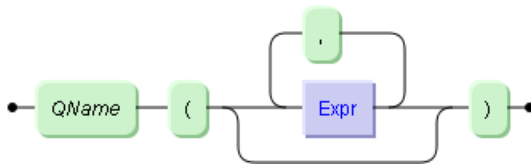
---

[XQueryModule](#)

## UpdatingFunction

Call a function to updated a database entry.

## Syntax



## Description

An `UpdatingFunction` represents the invocation of an updating function, together with the provided argument expression list. The `QName` must match the name of an updating function that is declared in the current update statement or in an imported module. The number of argument expressions must match the number of arguments in the declared function signature.

An updating function can only be invoked from within an update expression. It does not return a function value, rather it results in a sequence of update operations being applied to the database.

## Example

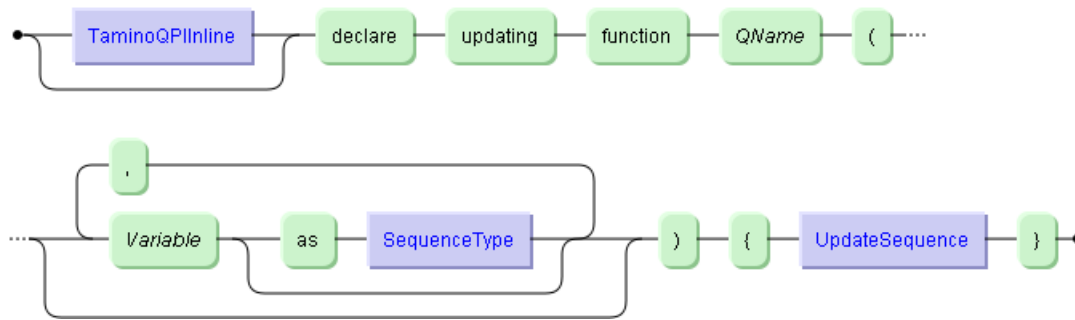
For each book element, invoke the function `y:set-year`:

```
import module namespace y="Y";
update
  for $book in input()/bib/book
  do Y:set-year($book)
```

## UpdatingFunctionDecl

Declare an updating function.

### Syntax



### Description

An `UpdatingFunctionDecl` declares an updating function. The declaration differs from the declaration of a non-updating function by the additional "updating" keyword, the absence of a result type specification, and an `UpdateSequence` instead of a result expression.

The `UpdatingFunctionDecl` occurs in the prolog of an update statement or a library module declaration. Although it is also syntactically valid in the prolog of a query, it is useless in queries because it can only be invoked from updating expressions.

### Example

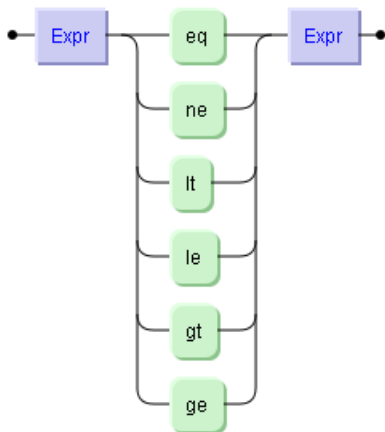
- This function inserts an attribute with the specified name into an element if it does not exist, or modifies the attribute value if it does exist.

```
declare updating function local:upsert($e as element(),
                                     $an as xs:QName,
                                     $av as xs:string)
{
  let $ea := $e/attribute()[node-name(.) = $an]
  do if ($ea) then
    replace $ea with attribute {$an} {$av}
  else
    insert attribute {$an} {$av} into $e
}
```

## ValueComp

Compare single values.

## Syntax



### ValueComp

## Description

A value comparison compares two single items. Each operand is atomized so that it is either a single atomic value or an empty sequence. If one of the operands is atomized to an empty sequence, the result of the operation is also an empty sequence. If the atomized operand is a sequence of length greater than one, an error is raised.

If one of the operands is of the type `xs:anySimpleType`, that operand is cast to a required type which is determined as follows:

- `xs:double`, if the type of the other operand is numeric
- `xs:string`, if the type of the other operand is `xs:anySimpleType`
- otherwise it is the type of the other operand

If the cast fails, a dynamic error is raised.

The result is `true` if the value of the first operand is equal, not equal, less than or equal, less, greater than, greater than or equal to the value of the second operand. Otherwise the result is `false`.

## Example

---

- Comparing two nodes with the same value, but different identities yields `true`:

```
<a>42</a> eq <a>42</a>
```

## Related Syntax Construct

---

[GeneralComp](#)



## VarDecl

Declare a variable in query prolog.

## Syntax



### VarDecl

## Description

A `VarDecl` declares a variable along with an optional type information in the prolog. It contains the keywords `declare variable`, followed by the variable name. An optional type information is introduced by the keyword `as`. The final keyword `external` is mandatory and indicates that the value of the variable will be provided by the external environment of the query. A type error is raised if in the dynamic phase of query processing the value is not of the specified type. If a variable declaration does not include a type declaration, then `item()*` is assumed.

Please note that a variable declaration differs from a variable binding as performed by `for` and `let` in FLWOR expressions. A variable declaration always refers to the declaration of a variable in a prolog, while a variable binding binds a variable to a value in a query expression and does not make the variable visible in the importing module.

## Example

- This variable declaration declares the variable `$x` as of type integer:

```
declare variable $x as xs:integer external
```

- This variable declaration declares the variable `$x`. As type `item()*` is assumed:

```
declare variable $x external
```

## Related Syntax Constructs

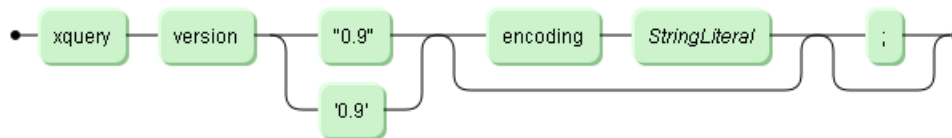
---

See also the discussion of external variables in the section [The `\_execute` command of the X-Machine programming documentation](#).

## VersionDecl

Specify the XQuery version to which the query conforms.

## Syntax



## Description

A `VersionDecl` specifies the version of XQuery to which the query conforms. It serves to select the language rules specific to that version, in order to provide selective support for multiple supported versions.

Though Tamino currently supports only a single dialect of XQuery, the version declaration has been added in preparation of future development. Tamino accepts version "0.9" as a specification of its XQuery dialect, and this is the only value currently permitted in the version declaration.

The version declaration can occur only at the very beginning of the statement. It should not even be preceded by any comments, because different versions will differ in the lexical structure of comments.

The optional specification of an encoding is ignored by the query processor. In fact the encoding of the query has already been resolved on request level.

## Example

```
xquery version "0.9";
```

## WhereClause

Filter intermediate results.

## Syntax

---



### WhereClause

## Description

---

A `WhereClause` is part of `FLWOR` and `FLWU` expressions and contains an expression. The `where` clause acts as a filter for the tuples previously generated by any `for` and `let` clauses. For each of these tuples the expression is evaluated. If the resulting effective boolean value is `true`, the tuple is retained and used in the `return` clause, otherwise the tuple is discarded.

## Example

---

See [FLWORExpr](#) for examples.

## Related Syntax Constructs

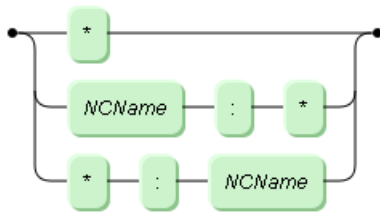
---

[ForClause](#)   [LetClause](#)

## Wildcard

Select nodes matching the principal node kind.

## Syntax



**Wildcard**

## Description

A Wildcard selects nodes of the principal node kind.

## Examples

- Retrieve Unicode transliterations for the current database:

```
declare namespace ino="http://namespaces.softwareag.com/tamino/response2"
declare namespace my_ino="http://namespaces.softwareag.com/tamino/response2"
input()//ino:transliteration/my_ino:*/@ino:*
```

Here two namespaces are declared and used in the query. The wildcard expression `my_ino:*` matches all nodes of the principal node kind element that are bound to the namespace prefixed by `my_ino`. The second expression `ino:@*` matches all nodes of the principal node kind attribute that are bound to the namespace prefixed by `ino`. Since both namespace prefixes bind the same namespace URI, the usage of `my_ino` and `ino` in this query is exchangeable.

- Select all names of the current collection:

```
input()//*:name
```

This query selects all `name` elements independent of the namespace.

## XmlProcessingInstruction

Construct an XML processing instruction.

### Syntax



**XmlProcessingInstruction**

### Description

An `XmlProcessingInstruction` directly constructs an XML processing instruction. It is delimited by "<?" and "?>" and contains a processing instruction target which is an XML name minus the letter sequence "XML" (regardless of case), see also the W3C specification of [PITarget](#). The `PITarget` must directly follow the opening delimiter "<?".

### Example

- This example constructs a processing instruction intended to be interpreted by the application `myapp`:

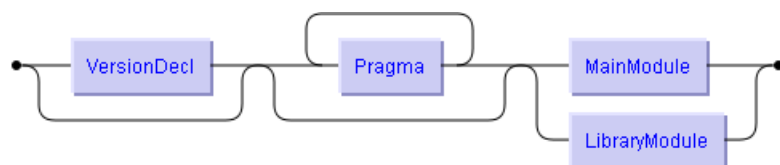
```
<?myapp rewrite="yes" ?>
```

## XQueryModule

Top-level XQuery syntax construct.

### Syntax

---



### XQueryModule

### Description

---

An `XQueryModule` is the top-level syntax construct in Tamino XQuery. A module is a fragment of XQuery code which either contains a prolog and the query body (main module) or it provides function and variable declarations (library module) that can be imported by other modules.

### Example

---

- Any query, even as simple as

```
input()
```

is formally a fragment of XQuery code. So it implies an XQuery main module that has no query prolog, but a query body that reads `input()`.



# III

## Functions

---

This chapter describes the functions available in Tamino XQuery 4. Wherever possible, the names correspond to the functions as defined in W3C XQuery. The links below lead to tables listing the available functions, along with the corresponding function in the W3C draft. The link in the W3C column leads you to the description in the W3C specification draft *XQuery 1.0 and XPath 2.0 Functions and Operators* of August 16, 2002. Remember that the *current* state of the W3C specification draft may look different. You can reference the functions in two ways:

- [Functions Ordered By Categories](#)
- [Functions in Alphabetical Order](#)

### Notation

---

The documentation of each function follows this pattern:

- A one-line summary;
- A section "Syntax" that specifies the function's signature(s);
- A section "Description" that documents usage and effect of the function;
- A section "Arguments" that describes the arguments of the function, if any;
- A section "Examples" showing different usage examples always using the same sample data.

The signature of a function looks like this:

```
function-name(argument-type $argument-name, ...) => return-type
```

Here, `function-name` is the name of the function, along with an optional namespace prefix. The prefix `xf` is used for functions that are bound to the default namespace <http://www.w3.org/2002/08/xquery-functions> and thus need not be explicitly declared. In parentheses, zero or more arguments can follow. The argument type is rendered in italics and the argument name is prefixed by `$`. The return type is at the end, also using italics.

# 5

## Functions Ordered by Categories

---

■ Tamino Functions .....	162
■ Tamino WebDAV Functions .....	162
■ Text Retrieval Functions .....	163
■ XQuery Functions .....	163

## Tamino Functions

---

The following functions are specific to Tamino XQuery 4. They are bound to the predeclared namespace *http://namespaces.softwareag.com/tamino/TaminoFunction* that is usually prefixed by `tf`.

Name	Short Description
<code>tf:content-type</code>	get the content type of the specified document
<code>tf:createTextNode</code>	create text node from string
<code>tf:document</code>	document constructor
<code>tf:getCollation</code>	get collation information for a single node
<code>tf:getCollection</code>	get name of current collection
<code>tf:get-current-user</code>	get name of current user
<code>tf:getDocname</code>	get name of current document
<code>tf:getInoId</code>	get <code>ino:id</code> of current document
<code>tf:getLastModified</code>	get date of last modification for current document
<code>tf:nonXML-kind</code>	get the kind of a non-XML document
<code>tf:parse</code>	convert non-XML content to XML
<code>tf:query</code>	Execute an XQuery
<code>tf:serialize</code>	convert XML content to non-XML
<code>tf:setDocname</code>	set name of current document
<code>tf:text-content</code>	retrieve non-XML text content from a non-XML document

## Tamino WebDAV Functions

---

The following functions are also specific to Tamino XQuery 4. They are bound to the namespace *http://namespaces.softwareag.com/tamino/DavFunction* that is usually prefixed by `tdf`.

Name	Short Description
<code>tdf:getProperties</code>	get all WebDAV properties of a document
<code>tdf:getProperty</code>	get single WebDAV property of a document
<code>tdf:isDescendantOf</code>	check for WebDAV resource on given path
<code>tdf:mkcol</code>	create a new collection resource
<code>tdf:resource</code>	get sequence of WebDAV resources
<code>tdf:setProperty</code>	set single WebDAV property of a document

## Text Retrieval Functions

The following functions perform operations in the context of text retrieval. They are also bound to the namespace *http://namespaces.softwareag.com/tamino/TaminoFunction* (prefixed by `tf`). Please note that the functions bound to the namespace *http://www.w3.org/2002/04/xquery-operators-text* (prefixed by `ft`) are deprecated and will be removed in a future version of Tamino.

Name	Short Description
<code>tf:broaderTerm</code>	search immediate superordinate term
<code>tf:broaderTerms</code>	search all superordinate terms
<code>tf:containsAdjacentText</code>	search word tokens in order within some distance
<code>tf:containsNearText</code>	search word tokens within some distance, but without order
<code>tf:containsText</code>	search for word tokens in a search string
<code>tf:content-type</code>	get content type of given document
<code>tf:createAdjacentTextReference</code>	create reference descriptions for adjacent text locations
<code>tf:createNearTextReference</code>	create reference descriptions for nearby text locations
<code>tf:createNodeReference</code>	create reference descriptions for nodes
<code>tf:createTextReference</code>	create reference descriptions for text locations
<code>tf:highlight</code>	highlight text based on reference descriptions
<code>tf:narrowerTerm</code>	search immediate subordinate term
<code>tf:narrowerTerms</code>	search all subordinate terms
<code>tf:phonetic</code>	search text based on phonetic similarities
<code>ft:proximity-contains</code>	search for word tokens within some distance
<code>tf:stem</code>	search text based on word stems
<code>tf:synonym</code>	search for synonymous terms
<code>ft:text-contains</code>	search for word tokens in a search string

## XQuery Functions

Most of the following functions are also defined in the W3C draft specification. They are bound to the namespace *http://www.w3.org/2002/08/xquery-functions* that is usually prefixed by `fn` in this documentation. Since this is the standard namespace in Tamino XQuery 4, you need not declare this namespace when invoking such a function. You can simply use `string-length` instead of `fn:string-length`.

## Accessors

Tamino XQuery	W3C XQuery Draft	Short Description
<code>fn:data</code>	<code>fn:data</code> (Draft of November 12, 2003)	return sequence of atomic items
<code>fn:node-name</code>	<code>fn:node-name</code> (Draft of October 29, 2004)	return expanded QName of a node
<code>fn:put</code>		inserts a new instance into a database
<code>fn:string</code>	<code>fn:string</code>	return argument value as a string

## Constructors

These functions construct an item of a supported type that is defined in the W3C XML Schema specification. This is why they are bound to the namespace <http://www.w3.org/2001/XMLSchema> and are usually prefixed by `xs`. Please also refer to the section [Constructor Functions](#) of the W3C draft specification *XQuery 1.0 and XPath 2.0 Functions and Operators* as of November 12, 2003.

Tamino XQuery	Short Description
<code>xs:anyURI</code>	construct an <code>anyURI</code> value from an item value
<code>xs:base64Binary</code>	construct a <code>base64Binary</code> value from an item value
<code>xs:boolean</code>	construct a <code>boolean</code> value from an item value
<code>xs:byte</code>	construct a <code>byte</code> value from an item value
<code>xs:date</code>	construct a <code>date</code> value from an item value
<code>xs:dateTime</code>	construct a <code>dateTime</code> value from an item value
<code>xs:decimal</code>	construct a <code>decimal</code> value from an item value
<code>xs:double</code>	construct a <code>double</code> value from an item value
<code>xs:duration</code>	construct a <code>duration</code> value from an item value
<code>xs:ENTITY</code>	construct an <code>ENTITY</code> value from an item value
<code>xs:float</code>	construct a <code>float</code> value from an item value
<code>xs:gDay</code>	construct a <code>gDay</code> value from an item value
<code>xs:gMonth</code>	construct a <code>gMonth</code> value from an item value
<code>xs:gMonthDay</code>	construct a <code>gMonthDay</code> value from an item value
<code>xs:gYear</code>	construct a <code>gYear</code> value from an item value
<code>xs:gYearMonth</code>	construct a <code>gYearMonth</code> value from an item value
<code>xs:hexBinary</code>	construct a <code>hexBinary</code> value from an item value
<code>xs:ID</code>	construct an <code>ID</code> value from an item value
<code>xs:IDREF</code>	construct an <code>IDREF</code> value from an item value
<code>xs:int</code>	construct an <code>int</code> value from an item value
<code>xs:integer</code>	construct an <code>integer</code> value from an item value
<code>xs:language</code>	construct a <code>language</code> value from an item value

Tamino XQuery	Short Description
<code>xs:long</code>	construct a <code>long</code> value from an item value
<code>xs:Name</code>	construct a <code>Name</code> value from an item value
<code>xs:NCName</code>	construct an <code>NCName</code> value from an item value
<code>xs:negativeInteger</code>	construct a <code>negativeInteger</code> value from an item value
<code>xs:NMTOKEN</code>	construct an <code>NMTOKEN</code> value from an item value
<code>xs:nonNegativeInteger</code>	construct a <code>nonNegativeInteger</code> value from an item value
<code>xs:nonPositiveInteger</code>	construct a <code>nonPositiveInteger</code> value from an item value
<code>xs:normalizedString</code>	construct a <code>normalizedString</code> value from an item value
<code>xs:positiveInteger</code>	construct a <code>positiveInteger</code> value from an item value
<code>xs:short</code>	construct a <code>short</code> value from an item value
<code>xs:string</code>	construct a <code>string</code> value from an item value
<code>xs:time</code>	construct a <code>time</code> value from an item value
<code>xs:token</code>	construct a <code>token</code> value from an item value
<code>xs:unsignedByte</code>	construct an <code>unsignedByte</code> value from an item value
<code>xs:unsignedInt</code>	construct an <code>unsignedInt</code> value from an item value
<code>xs:unsignedLong</code>	construct an <code>unsignedLong</code> value from an item value
<code>xs:unsignedShort</code>	construct an <code>unsignedShort</code> value from an item value

## Functions on Numbers

Tamino XQuery	W3C XQuery Draft	Short Description
<code>fn:abs</code>		return absolute value
<code>fn:ceiling</code>	<code>fn:ceiling</code>	return smallest integer not smaller than its argument
<code>fn:floor</code>	<code>fn:floor</code>	return largest integer not greater than its argument
<code>fn:round</code>	<code>fn:round</code>	return number closest to its argument

## Functions on Strings

Tamino XQuery	W3C XQuery Draft	Short Description
<code>fn:compare</code>	<code>fn:compare</code> ( <i>Draft of November 12, 2003</i> )	compare two strings
<code>fn:contains</code>	<code>fn:contains</code> ( <i>Draft of November 12, 2003</i> )	check whether one string contains another
<code>fn:ends-with</code>	<code>fn:ends-with</code> ( <i>Draft of November 12, 2003</i> )	check whether a string ends with another string
<code>fn:lower-case</code>	<code>fn:lower-case</code> ( <i>Draft of October 29, 2004</i> )	return lower-cased value of a string

Tamino XQuery	W3C XQuery Draft	Short Description
<code>fn:normalize-space</code>	<code>fn:normalize-space</code>	return its argument with normalized whitespace
<code>fn:starts-with</code>	<code>fn:starts-with</code>	check whether string 1 starts with string 2
<code>fn:string-join</code>	<code>fn:string-join</code> (Draft of November 12, 2003)	return concatenation of string sequence
<code>fn:string-length</code>	<code>fn:string-length</code>	return length of string value
<code>fn:substring</code>	<code>fn:substring</code> (Draft of November 12, 2003)	return substring of a string value
<code>fn:substring-after</code>	<code>fn:substring-after</code> (Draft of November 12, 2003)	return substring of a string value following another string
<code>fn:substring-before</code>	<code>fn:substring-before</code> (Draft of November 12, 2003)	return substring of a string value preceding another string
<code>fn:upper-case</code>	<code>fn:upper-case</code> (Draft of October 29, 2004)	return upper-cased value of a string

## Functions on Booleans

Tamino XQuery	W3C XQuery Draft	Short Description
<code>fn:false</code>	<code>fn:false</code>	return the boolean value <code>false</code>
<code>fn:not</code>	<code>fn:not</code>	invert boolean value of its argument
<code>fn:true</code>	<code>fn:true</code>	return the boolean value <code>true</code>

## Component Extraction Functions on Date and Time Values

These functions are all based on the W3C draft specification *XQuery 1.0 and XPath 2.0 Functions and Operators* as of October 29, 2004.

Tamino XQuery	W3C XQuery Draft	Short Description
<code>fn:day-from-date</code>	<code>fn:day-from-date</code>	return day integer value from date argument
<code>fn:day-from-dateTime</code>	<code>fn:day-from-dateTime</code>	return day integer value from datetime argument
<code>fn:hours-from-dateTime</code>	<code>fn:hours-from-dateTime</code>	return hours integer value from datetime argument
<code>fn:hours-from-time</code>	<code>fn:hours-from-time</code>	return hours integer value from time argument
<code>fn:minutes-from-dateTime</code>	<code>fn:minutes-from-dateTime</code>	return minutes integer value from datetime argument



Tamino XQuery	W3C XQuery Draft	Short Description
<code>fn:minutes-from-time</code>	<code>fn:minutes-from-time</code>	return minutes integer value from time argument
<code>fn:month-from-date</code>	<code>fn:month-from-date</code>	return month integer value from date argument
<code>fn:month-from-dateTime</code>	<code>fn:month-from-dateTime</code>	return month integer value from datetime argument
<code>fn:seconds-from-dateTime</code>	<code>fn:seconds-from-dateTime</code>	return seconds integer value from datetime argument
<code>fn:seconds-from-time</code>	<code>fn:seconds-from-time</code>	return seconds integer value from time argument
<code>fn:year-from-date</code>	<code>fn:year-from-date</code>	return year integer value from date argument
<code>fn:year-from-dateTime</code>	<code>fn:year-from-dateTime</code>	return year integer value from datetime argument

## Functions on QNames

Tamino XQuery	W3C XQuery Draft	Short Description
<code>fn:expanded-QName</code>	<code>fn:expanded-QName</code>	return a constructed QName
<code>fn:get-local-name-from-QName</code>	<code>fn:get-local-name-from-QName</code>	return local part of QName argument
<code>fn:get-namespace-from-QName</code>	<code>fn:get-namespace-from-QName</code>	return namespace URI of QName argument

## Functions on Nodes

Tamino XQuery	W3C XQuery Draft	Short Description
<code>fn:local-name</code>	<code>fn:local-name</code>	return local name of node
<code>fn:namespace-uri</code>	<code>fn:namespace-uri</code>	return namespace URI from node
<code>fn:root</code>	<code>fn:root</code>	return root of tree with argument node

## Functions on Sequences

Tamino XQuery	W3C XQuery Draft	Short Description
<code>fn:avg</code>	<code>fn:avg</code>	return average of a sequence of numbers
<code>fn:boolean</code>	<code>fn:boolean</code> (Draft of November 12, 2003)	compute effective Boolean value of a sequence of items
<code>fn:count</code>	<code>fn:count</code>	return number of items in the argument's value.
<code>fn:deep-equal</code>	<code>fn:deep-equal</code> (Draft of October 29, 2004)	check for items in two arguments that compare equal in corresponding positions
<code>fn:distinct-values</code>	<code>fn:distinct-values</code> (Draft of November 12, 2003)	return sequence with distinct values
<code>fn:max</code>	<code>fn:max</code>	return object with maximum value from item sequence
<code>fn:min</code>	<code>fn:min</code>	return object with minimum value from item sequence
<code>fn:reverse</code>	<code>fn:reverse</code>	reverse the order of items in a sequence
<code>fn:subsequence</code>	<code>fn:subsequence</code>	return subsequence of a sequence
<code>fn:sum</code>	<code>fn:sum</code>	return sum of a sequence of numbers

## Functions that Generate Sequences

Tamino XQuery	W3C XQuery Draft	Short Description
<code>fn:collection</code>	<code>fn:collection</code> (Draft of November 12, 2003)	return input sequence from specified collection
<code>fn:id</code>	<code>fn:id</code> (Draft of October 29, 2004)	return sequence of element nodes referenced by IDREF values
<code>fn:idref</code>	<code>fn:idref</code> (Draft of October 29, 2004)	return sequence of attribute nodes with IDREF values containing at least a specified ID value

## Context Functions

Tamino XQuery	W3C XQuery Draft	Short Description
<code>fn:current-date</code>	<code>fn:current-date</code> (Draft of October 29, 2004)	return current date
<code>fn:current-dateTime</code>	<code>fn:current-dateTime</code> (Draft of October 29, 2004)	return current date and time
<code>fn:current-time</code>	<code>fn:current-time</code> (Draft of October 29, 2004)	return current time
<code>fn:last</code>	<code>fn:last</code>	return number of items in the current sequence

Tamino XQuery	W3C XQuery Draft	Short Description
<code>fn:position</code>	<code>fn:position</code>	return position of context item in current sequence



## 6 Functions in Alphabetical Order

---

▪ fn:abs .....	183
▪ fn:avg .....	184
▪ fn:boolean .....	185
▪ fn:ceiling .....	186
▪ fn:collection .....	188
▪ fn:compare .....	189
▪ fn:concat .....	191
▪ fn:contains .....	192
▪ fn:count .....	194
▪ fn:current-date .....	195
▪ fn:current-dateTime .....	196
▪ fn:current-time .....	197
▪ fn:data .....	198
▪ fn:day-from-date .....	199
▪ fn:day-from-dateTime .....	200
▪ fn:deep-equal .....	201
▪ fn:distinct-values .....	204
▪ fn:ends-with .....	206
▪ fn:expanded-QName .....	208
▪ fn:false .....	209
▪ fn:floor .....	210
▪ fn:get-local-name-from-QName .....	212
▪ fn:get-namespace-from-QName .....	213
▪ fn:hours-from-dateTime .....	214
▪ fn:hours-from-time .....	215
▪ fn:id .....	216
▪ fn:idref .....	217
▪ fn:last .....	218
▪ fn:local-name .....	219
▪ fn:lower-case .....	220
▪ fn:matches .....	221
▪ fn:max .....	223

---

▪ fn:min .....	224
▪ fn:minutes-from-dateTime .....	225
▪ fn:minutes-from-time .....	226
▪ fn:month-from-date .....	227
▪ fn:month-from-dateTime .....	228
▪ fn:namespace-uri .....	229
▪ fn:node-name .....	230
▪ fn:normalize-space .....	231
▪ fn:not .....	232
▪ fn:position .....	233
▪ fn:put .....	234
▪ fn:replace .....	236
▪ fn:reverse .....	238
▪ fn:root .....	239
▪ fn:round .....	240
▪ fn:seconds-from-dateTime .....	242
▪ fn:seconds-from-time .....	243
▪ fn:starts-with .....	244
▪ fn:string .....	245
▪ fn:string-join .....	247
▪ fn:string-length .....	249
▪ fn:subsequence .....	250
▪ fn:substring .....	252
▪ fn:substring-after .....	254
▪ fn:substring-before .....	256
▪ fn:sum .....	258
▪ fn:tokenize .....	259
▪ fn:true .....	260
▪ fn:upper-case .....	261
▪ fn:year-from-date .....	262
▪ fn:year-from-dateTime .....	263
▪ ft:proximity-contains .....	264
▪ ft:text-contains .....	267
▪ tdf:getProperties .....	269
▪ tdf:getProperty .....	270
▪ tdf:isDescendantOf .....	271
▪ tdf:mkcol .....	273
▪ tdf:resource .....	274
▪ tdf:setProperty .....	275
▪ tf:broaderTerm .....	276
▪ tf:broaderTerms .....	278
▪ tf:containsAdjacentText .....	280
▪ tf:containsNearText .....	282
▪ tf:containsText .....	284
▪ tf:content-type .....	286

▪ tf:createAdjacentTextReference .....	287
▪ tf:createNearTextReference .....	289
▪ tf:createNodeReference .....	291
▪ tf:createTextNode .....	292
▪ tf:createTextReference .....	294
▪ tf:document .....	297
▪ tf:getCollation .....	298
▪ tf:getCollection .....	300
▪ tf:get-current-user .....	301
▪ tf:getDocname .....	302
▪ tf:getInold .....	303
▪ tf:getLastModified .....	304
▪ tf:highlight .....	306
▪ tf:narrowerTerm .....	308
▪ tf:narrowerTerms .....	310
▪ tf:nonXML-kind .....	312
▪ tf:parse .....	313
▪ tf:phonetic .....	314
▪ tf:query .....	316
▪ tf:serialize .....	317
▪ tf:setDocname .....	318
▪ tf:stem .....	319
▪ tf:synonym .....	321
▪ tf:text-content .....	323
▪ xs:anyURI .....	324
▪ xs:base64Binary .....	325
▪ xs:boolean .....	326
▪ xs:byte .....	328
▪ xs:date .....	330
▪ xs:dateTime .....	332
▪ xs:decimal .....	334
▪ xs:double .....	335
▪ xs:duration .....	336
▪ xs:ENTITY .....	338
▪ xs:float .....	340
▪ xs:gDay .....	341
▪ xs:gMonth .....	342
▪ xs:gMonthDay .....	343
▪ xs:gYear .....	344
▪ xs:gYearMonth .....	346
▪ xs:hexBinary .....	348
▪ xs:ID .....	349
▪ xs:IDREF .....	351
▪ xs:int .....	353
▪ xs:integer .....	354

■ xs:language .....	355
■ xs:long .....	356
■ xs:Name .....	358
■ xs:NCName .....	360
■ xs:NMTOKEN .....	362
■ xs:negativeInteger .....	364
■ xs:nonNegativeInteger .....	366
■ xs:nonPositiveInteger .....	368
■ xs:normalizedString .....	370
■ xs:positiveInteger .....	371
■ xs:short .....	373
■ xs:string .....	375
■ xs:time .....	376
■ xs:token .....	377
■ xs:unsignedByte .....	378
■ xs:unsignedInt .....	380
■ xs:unsignedLong .....	382
■ xs:unsignedShort .....	384

XQuery 4 Function	W3C Function	Category	Description
<code>fn:abs</code>	—	numeric	return absolute value of an item
<code>fn:avg</code>	<code>fn:avg</code>	aggregate	return average of a sequence of numbers
<code>fn:boolean</code>	<code>fn:boolean</code> ( <i>Draft of November 12, 2003</i> )	sequence	compute effective Boolean value of a sequence of items
<code>fn:ceiling</code>	<code>fn:ceiling</code>	numeric	return smallest integer not smaller than its argument
<code>fn:collection</code>	<code>fn:collection</code> ( <i>Draft of November 12, 2003</i> )	sequence-generating	return input sequence from specified collection
<code>fn:compare</code>	<code>fn:compare</code>	string	compare two strings
<code>fn:concat</code>	<code>fn:concat</code>	string	concatenates the values of its arguments
<code>fn:contains</code>	<code>fn:contains</code>	string	check whether one string contains another
<code>fn:count</code>	<code>fn:count</code>	aggregate	return number of items in the argument's value
<code>fn:current-date</code>	<code>fn:current-date</code> ( <i>Draft of October 29, 2004</i> )	context	return current date



XQuery 4 Function	W3C Function	Category	Description
<code>fn:current-dateTime</code>	<code>fn:current-dateTime</code> (Draft of October 29, 2004)	context	return current date time
<code>fn:current-time</code>	<code>fn:current-time</code> (Draft of October 29, 2004)	context	return current time
<code>fn:data</code>	<code>fn:data</code> (Draft of November 12, 2003)	accessor	return sequence of atomic items
<code>fn:day-from-date</code>	<code>fn:day-from-date</code> (Draft of October 29, 2004)	date	return day integer value from date argument
<code>fn:day-from-dateTime</code>	<code>fn:day-from-dateTime</code> (Draft of October 29, 2004)	date	return day integer value from datetime argument
<code>fn:deep-equal</code>	<code>fn:deep-equal</code> (Draft of October 29, 2004)	sequence	check for items in two arguments that compare equal in corresponding positions
<code>fn:distinct-values</code>	<code>fn:distinct-values</code> (Draft of November 12, 2003)	sequence	return sequence with distinct values
<code>fn:ends-with</code>	<code>fn:ends-with</code>	string	check whether a string ends with another string
<code>fn:expanded-QName</code>	<code>fn:expanded-QName</code>	QName	return a constructed QName
<code>fn:false</code>	<code>fn:false</code>	boolean	return the boolean value <code>false</code>
<code>fn:floor</code>	<code>fn:floor</code>	numeric	return largest integer not greater than its argument
<code>fn:get-local-name-from-QName</code>	<code>fn:get-local-name-from-QName</code>	QName	return local part of QName argument
<code>fn:get-namespace-from-QName</code>	<code>fn:get-namespace-from-QName</code>	QName	return namespace URI of QName argument
<code>fn:hours-from-dateTime</code>	<code>fn:hours-from-dateTime</code> (Draft of October 29, 2004)	date	return hours integer value from datetime argument
<code>fn:hours-from-time</code>	<code>fn:hours-from-time</code> (Draft of October 29, 2004)	date	return hours integer value from time argument
<code>fn:id</code>	<code>fn:id</code> (Draft of October 29, 2004)	sequence-generating	return sequence of element nodes referenced by IDREF values

XQuery 4 Function	W3C Function	Category	Description
<code>fn:idref</code>	<code>fn:idref</code> ( <i>Draft of October 29, 2004</i> )	sequence-generating	return sequence of attribute nodes with IDREF values containing at least a specified ID value
<code>fn:last</code>	<code>fn:last</code>	context	return number of items in the current sequence
<code>fn:local-name</code>	<code>fn:local-name</code>	node	return local name of node
<code>fn:lower-case</code>	<code>fn:lower-case</code> ( <i>Draft of November 12, 2003</i> )	string	return lower-cased value of a string
<code>fn:matches</code>	<code>fn:matches</code>	patternmatching	patternmatching
<code>fn:max</code>	<code>fn:max</code>	aggregate	return object with maximum value from item sequence
<code>fn:min</code>	<code>fn:min</code>	aggregate	return object with minimum value from item sequence
<code>fn:minutes-from-dateTime</code>	<code>fn:minutes-from-dateTime</code> ( <i>Draft of October 29, 2004</i> )	date	return minutes integer value from datetime argument
<code>fn:minutes-from-time</code>	<code>fn:minutes-from-time</code> ( <i>Draft of October 29, 2004</i> )	date	return minutes integer value from time argument
<code>fn:month-from-date</code>	<code>fn:month-from-date</code> ( <i>Draft of October 29, 2004</i> )	date	return month integer value from date argument
<code>fn:month-from-dateTime</code>	<code>fn:month-from-dateTime</code> ( <i>Draft of October 29, 2004</i> )	date	return month integer value from datetime argument
<code>fn:namespace-uri</code>	<code>fn:namespace-uri</code>	node	return namespace URI from node
<code>fn:node-name</code>	<code>fn:node-name</code> ( <i>Draft of October 29, 2004</i> )	node	return expanded QName of a node
<code>fn:normalize-space</code>	<code>fn:normalize-space</code>	string	return its argument with normalized whitespace
<code>fn:not</code>	<code>fn:not</code>	boolean	invert boolean value of its argument
<code>fn:position</code>	<code>fn:position</code>	context	return position of context item in current sequence

XQuery 4 Function	W3C Function	Category	Description
<code>fn:put</code>	<code>fn:put</code>		inserts new instance into database
<code>fn:replace</code>	<code>fn:replace</code>	patternmatching	patternmatching
<code>fn:reverse</code>	<code>fn:reverse</code>	node	reverses the order of items in a sequence
<code>fn:root</code>	<code>fn:root</code>	node	return root of tree w argument node
<code>fn:round</code>	<code>fn:round</code>	numeric	return number close to its argument
<code>fn:seconds-from-dateTime</code>	<code>fn:seconds-from-dateTime</code> (Draft of October 29, 2004)	date	return seconds interval value from datetime argument
<code>fn:seconds-from-time</code>	<code>fn:seconds-from-time</code> (Draft of October 29, 2004)	date	return seconds interval value from time argument
<code>fn:starts-with</code>	<code>fn:starts-with</code> (Draft of November 12, 2003)	string	check whether string starts with string 2
<code>fn:string</code>	<code>fn:string</code>	accessor	return argument value as a string
<code>fn:string-join</code>	<code>fn:string-join</code> (Draft of November 12, 2003)	string	return concatenation of string sequence
<code>fn:string-length</code>	<code>fn:string-length</code>	string	return length of string value
<code>fn:subsequence</code>	<code>fn:subsequence</code>	sequence	return subsequence of a sequence
<code>fn:substring</code>	<code>fn:substring</code> (Draft of November 12, 2003)	string	return substring of string value
<code>fn:substring-after</code>	<code>fn:substring-after</code> (Draft of November 12, 2003)	string	return substring of string value following another string
<code>fn:substring-before</code>	<code>fn:substring-before</code> (Draft of November 12, 2003)	string	return substring of string value preceding another string
<code>fn:sum</code>	<code>fn:sum</code>	aggregate	return sum of a sequence of numbers
<code>fn:tokenize</code>	<code>fn:tokenize</code>	patternmatching	patternmatching
<code>fn:true</code>	<code>fn:true</code>	boolean	return the boolean value true
<code>fn:upper-case</code>	<code>fn:upper-case</code> (Draft of November 12, 2003)	string	return upper-cased value of a string

XQuery 4 Function	W3C Function	Category	Description
<code>fn:year-from-date</code>	<code>fn:year-from-date</code> ( <i>Draft of October 29, 2004</i> )	date	return year integer value from date argument
<code>fn:year-from-dateTime</code>	<code>fn:year-from-dateTime</code> ( <i>Draft of October 29, 2004</i> )	date	return year integer value from datetime argument
<code>ft:proximity-contains</code>	—	text retrieval	search for word tokens within some distance
<code>ft:text-contains</code>	—	text retrieval	search for word tokens in a search string
<code>tdf:getProperties</code>	—	WebDAV	get all WebDAV properties of a document
<code>tdf:getProperty</code>	—	WebDAV	get single WebDAV property of a document
<code>tdf:isDescendantOf</code>	—	WebDAV	check for WebDAV resource on given path
<code>tdf:mkcol</code>	—	WebDAV	create a new collection resource
<code>tdf:resource</code>	—	WebDAV	get sequence of WebDAV resources
<code>tdf:setProperty</code>	—	WebDAV	set single WebDAV property of a document
<code>tf:broaderTerm</code>	—	text retrieval	search for immediate superordinate term
<code>tf:broaderTerms</code>	—	text retrieval	search for all superordinate terms
<code>tf:containsAdjacentText</code>	—	text retrieval	search for word tokens in order within some distance
<code>tf:containsNearText</code>	—	text retrieval	search for word tokens within some distance, but without order
<code>tf:containsText</code>	—	text retrieval	search for word tokens in a search string
<code>tf:content-type</code>	—		get content type of given document
<code>tf:createAdjacentTextReference</code>	—	text retrieval	create reference descriptions for adjacent text locations

XQuery 4 Function	W3C Function	Category	Description
<code>tf:createNearTextReference</code>	—	text retrieval	create reference descriptions for near text locations
<code>tf:createNodeReference</code>	—	text retrieval	create reference descriptions for nodes
<code>tf:createTextNode</code>	—	Tamino	create a text node from a string
<code>tf:createTextReference</code>	—	text retrieval	create reference descriptions for text locations
<code>tf:document</code>			construct a document
<code>tf:getCollation</code>	—	Tamino	get collation information for a string node
<code>tf:getCollection</code>	—	Tamino	get name of current collection
<code>tf:get-current-user</code>	—	Tamino	get name of current user
<code>tf:getDocname</code>	—	Tamino	get name of current document
<code>tf:getInoId</code>	—	Tamino	get <code>ino:id</code> of current document
<code>tf:getLastModified</code>	—	Tamino	get date of last modification for current document
<code>tf:highlight</code>	—	text retrieval	highlight text based on reference description
<code>tf:narrowerTerm</code>	—	text retrieval	search for immediate subordinate term
<code>tf:narrowerTerms</code>	—	text retrieval	search for all subordinate terms
<code>tf:nonXML-kind</code>			get the kind (text or binary) of a non-XML document
<code>tf:parse</code>			convert non-XML content to XML
<code>tf:phonetic</code>	—	text retrieval	search text based on phonetic similarities
<code>tf:query</code>			Execute an XQuery
<code>tf:serialize</code>			convert XML content to non-XML

XQuery 4 Function	W3C Function	Category	Description
<code>tf:setDocname</code>	—	Tamino	modify name of a document
<code>tf:stem</code>	—	text retrieval	search text based on word stems
<code>tf:synonym</code>	—	text retrieval	search for synonymous terms
<code>tf:text-content</code>			get non-XML text content
<code>xs:anyURI</code>	<code>xs:anyURI</code>	constructor	construct an <code>anyURI</code> value from an item value
<code>xs:base64Binary</code>	<code>xs:base64Binary</code>	constructor	construct a <code>base64Binary</code> value from an item value
<code>xs:boolean</code>	<code>xs:boolean</code>	constructor	construct a <code>boolean</code> value from an item value
<code>xs:byte</code>	<code>xs:byte</code>	constructor	construct a <code>byte</code> value from an item value
<code>xs:date</code>	<code>xs:date</code>	constructor	construct a <code>date</code> value from an item value
<code>xs:dateTime</code>	<code>xs:dateTime</code>	constructor	construct a <code>dateTime</code> value from an item value
<code>xs:decimal</code>	<code>xs:decimal</code>	constructor	construct a <code>decimal</code> value from an item value
<code>xs:double</code>	<code>xs:double</code>	constructor	construct a <code>double</code> value from an item value
<code>xs:duration</code>	<code>xs:duration</code>	constructor	construct a <code>duration</code> value from an item value
<code>xs:ENTITY</code>	<code>xs:ENTITY</code>	constructor	construct an <code>ENTITY</code> value from an item value
<code>xs:float</code>	<code>xs:float</code>	constructor	construct a <code>float</code> value from an item value
<code>xs:gDay</code>	<code>xs:gDay</code>	constructor	construct a <code>gDay</code> value from an item value

XQuery 4 Function	W3C Function	Category	Description
<code>xs:gMonth</code>	<code>xs:gMonth</code>	constructor	construct a <code>gMonth</code> value from an item value
<code>xs:gMonthDay</code>	<code>xs:gMonthDay</code>	constructor	construct a <code>gMonthDay</code> value from an item value
<code>xs:gYear</code>	<code>xs:gYear</code>	constructor	construct a <code>gYear</code> value from an item value
<code>xs:gYearMonth</code>	<code>xs:gYearMonth</code>	constructor	construct a <code>gYearMonth</code> value from an item value
<code>xs:hexBinary</code>	<code>xs:hexBinary</code>	constructor	construct a <code>hexBinary</code> value from an item value
<code>xs:ID</code>	<code>xs:ID</code>	constructor	construct an <code>ID</code> value from an item value
<code>xs:IDREF</code>	<code>xs:IDREF</code>	constructor	construct an <code>IDREF</code> value from an item value
<code>xs:int</code>	<code>xs:int</code>	constructor	construct an <code>int</code> value from an item value
<code>xs:integer</code>	<code>xs:integer</code>	constructor	construct an <code>integer</code> value from an item value
<code>xs:language</code>	<code>xs:language</code>	constructor	construct a <code>language</code> value from an item value
<code>xs:long</code>	<code>xs:long</code>	constructor	construct a <code>long</code> value from an item value
<code>xs:name</code>	<code>xs:Name</code>	constructor	construct a <code>Name</code> value from an item value
<code>xs:NCName</code>	<code>xs:NCName</code>	constructor	construct an <code>NCName</code> value from an item value
<code>xs:negativeInteger</code>	<code>xs:negativeInteger</code>	constructor	construct a <code>negativeInteger</code> value from an item value
<code>xs:NMTOKEN</code>	<code>xs:NMTOKEN</code>	constructor	construct an <code>NMTOKEN</code> value from an item value
<code>xs:nonNegativeInteger</code>	<code>xs:nonNegativeInteger</code>	constructor	construct a <code>nonNegativeInteger</code> value from an item value

XQuery 4 Function	W3C Function	Category	Description
			value from an item value
<code>xs:nonPositiveInteger</code>	<code>xs:nonPositiveInteger</code>	constructor	construct a <code>nonPositiveInteger</code> value from an item value
<code>xs:normalizedString</code>	<code>xs:normalizedString</code>	constructor	construct a <code>normalizedString</code> value from an item value
<code>xs:positiveInteger</code>	<code>xs:positiveInteger</code>	constructor	construct a <code>positiveInteger</code> value from an item value
<code>xs:short</code>	<code>xs:short</code>	constructor	construct a <code>short</code> value from an item value
<code>xs:string</code>	<code>xs:string</code>	constructor	construct a <code>string</code> value from an item value
<code>xs:time</code>	<code>xs:time</code>	constructor	construct a <code>time</code> value from an item value
<code>xs:token</code>	<code>xs:token</code>	constructor	construct a <code>token</code> value from an item value
<code>xs:unsignedByte</code>	<code>xs:unsignedByte</code>	constructor	construct a <code>unsignedByte</code> value from an item value
<code>xs:unsignedInt</code>	<code>xs:unsignedInt</code>	constructor	construct a <code>unsignedInt</code> value from an item value
<code>xs:unsignedLong</code>	<code>xs:unsignedLong</code>	constructor	construct a <code>unsignedLong</code> value from an item value
<code>xs:unsignedShort</code>	<code>xs:unsignedShort</code>	constructor	construct a <code>unsignedShort</code> value from an item value



## fn:abs

Return absolute value of an item.

### Syntax

---

```
fn:abs(item? $value) => double
```

### Description

---

This function returns the absolute value of an item. The item must evaluate to a number.

### Argument

---

**\$number**  
item?

### Example

---

```
let $a := xs:double(17.5)  
return abs($a)
```

## fn:avg

Return average of a sequence of numbers.

### Syntax

---

```
fn:avg(sequence $sequence) => double
```

### Description

---

This aggregate function returns the average of the numeric values of the nodes of `$sequence`. The nodes of the sequence must all evaluate to numbers. If a node evaluates to the empty sequence it is discarded.

### Argument

---

**\$sequence**  
sequence of nodes

### Example

---

```
let $a := (17, 23, 11)
return avg($a)
```

## fn:boolean

Compute effective Boolean value of a sequence of items.

### Syntax

```
fn:boolean(item* $srcval) => boolean
```

### Description

This function returns the effective Boolean value of a sequence of items. If the sequence is empty, the function returns "false". If `$srcval` is an atomic value, this function returns the same value as the constructor function `xs:boolean`. In the following cases the function returns "false":

- The singleton Boolean value "false";
- A singleton numeric value that is numerically equal to zero;
- The empty string value "".

### Argument

**\$srcval**  
any item

### Example

```
boolean(24-24)  
boolean("24-24")
```

The first query returns "false", since the argument is a numerical expression that evaluates to zero. In the second query the argument is a non-empty string value, and therefore the function returns "true".

## fn:ceiling

Return smallest integer not smaller than its argument.

### Syntax

---

```
fn:ceiling(double? $number) => double?
```

### Description

---

This function returns the smallest integer that is not smaller than the value of `$number`. If the argument is the empty sequence, it returns the empty sequence. If the argument is a literal, then the literal must be a valid lexical form of its type.

### Argument

---

**\$number**

double-precision number

### Examples

---

- Call function with the numeric value 41.7:

```
ceiling(41.7E0)
```

```
ceiling(xs:double("41.7"))
```

In both cases the double-precision number 4.2E1 is returned. In the first query the argument is a valid literal of the required type, in the second query the required type is constructed using the constructor function `xs:double`.

- Call function with the numeric value -273.15:

```
ceiling(-273.15E0)
```

This query returns -2.73E2.

## fn:collection

Return an input sequence from a specified collection.

### Syntax

---

```
fn:collection(string $srcval) => node*
```

### Description

---

This function returns a sequence of document nodes of all documents in a Tamino collection. The collection must be defined in the current database.

### Argument

---

#### **\$srcval**

The name of the collection in the current database. It must be a string literal.

### Example

---

- Get the characters that are used as delimiter characters in the current database:

```
declare namespace ino="http://namespaces.softwareag.com/tamino/response2"
for   $a in collection("ino:vocabulary")/ino:transliteration/ino:character
where $a/@ino:class eq "delimiter"
return $a
```

## fn:compare

Compare two strings.

### Syntax

---

```
fn:compare(string $string1, string? $string2) => integer
```

```
fn:compare(string $string1, string? $string2, string $collation) => integer
```

### Description

---

This function compares two strings. If a third parameter is supplied, the strings are compared according to the specified collation. If the third parameter is not supplied, the strings are compared according to the default collation. It returns one of the following integer values:

- 1 the value of `$string1` is less than that of `$string2`
- 0 the value of `$string1` is equal to that of `$string2`
- 1 the value of `$string1` is greater than that of `$string2`

If no collation is used, Unicode code points are used when comparing.

### Arguments

---

**\$string1**  
string value

**\$string2**  
string value

**\$collation**  
string value `$string`

### Examples

---

- Compare two strings that differ in case:

```
compare("Tamino", "tamino")
```

This query returns -1, since the value of "T" (U+0054) is less than that of "t" (U+0074).

- Perform a case-insensitive comparison:

```
compare("Tamino", "tamino", "collation?strength=secondary")
```

This query returns 0, since the values of the strings "Tamino" and "tamino", disregarding upper-/lower-case, are equal. The collation strength "secondary" indicates that primary character attributes, i.e. the base character itself, and secondary attributes, i.e. accents, are considered when performing the comparison, but tertiary and quaternary attributes, which include case information, are ignored. See `tsd:strength` for further information.

- Get all patients with the name "Müller":

```
for $i in input()/patient/name
where compare($i/surname, 'Müller', tf:getCollation($i/surname)) = 0
return $i
```

This query returns the `name` nodes of all patients whose `surname` contains the string value "Müller" or a string value that is equal to "Müller" using the collation defined for the element `surname`.



## fn:concat

Returns the concatenation of the values of its arguments.

### Syntax

```
fn:concat() => string
```

```
fn:concat(string? $string1) => string
```

```
fn:concat(string? $string1, string? $string2, ...) => string
```

### Description

This function concatenates the values of its arguments. It accepts zero or more `xs:string`s as arguments. In case of no arguments, the empty string is returned.

### Arguments

**\$string1, \$string2, ...**  
string value

### Examples

```
■ concat("a","b")
```

This returns "ab".

## fn:contains

Check whether one string contains another.

### Syntax

---

```
fn:contains(string $string, string? $searchString) => boolean
```

```
fn:contains(string $string, string? $searchString, string $collation) => boolean
```

### Description

---

This function checks whether the value of `$string` contains the value of `$searchString` anywhere. It returns `true` if it is found and `false` if not. There are some special cases in which this function returns the following values:

`true` the value of `$string` is a string of length zero  
`false` the value of `$searchString` is a string of length zero

It is an error if either string value is an empty sequence.

### Arguments

---

**\$string**  
string value

**\$searchString**  
string value that is searched for in `$string`

**\$collation**  
optional valid collation string literal

### Examples

---

- Is "ino" in "Tamino"?

```
contains("Tamino", "ino")
```

The function returns `true`, since "ino" is obviously contained in "Tamino".

- Is the empty string part of another empty string?

```
contains("", "")
```

The function returns `true`.

## fn:count

Return number of items in the argument's value.

### Syntax

---

```
fn:count(item* $sequence) => unsignedInt
```

### Description

---

This aggregate function returns the number of items in the value of `$sequence`. If `$sequence` is the empty sequence, the function returns zero.

### Argument

---

**\$sequence**  
item value

### Example

---

- Count all comment nodes in the current collection:

```
count(input()//comment())
```

## fn:current-date

Return current date.

### Syntax

---

```
fn:current-date() => xs:date
```

### Description

---

This function returns the current date corresponding to the current date and time. The result is a value of type `xs:date` that contains the date together with timezone information.

### Examples

---

- Calling this function can return a value such as 2005-09-25Z:

```
current-date()
```

## fn:current-dateTime

Return the current date and time.

### Syntax

---

```
fn:current-dateTime() => xs:dateTime
```

### Description

---

This function returns a value of type `xs:dateTime` corresponding to the current date and time. The result is a value of type `xs:dateTime` that contains the date and time together with timezone information.

### Examples

---

- Calling this function can return a value such as `2005-09-25T17:22:33.161Z` meaning the current time on September 25, 2005:

```
current-dateTime()
```

## fn:current-time

Return current time.

### Syntax

---

```
fn:current-time() => xs:time
```

### Description

---

This function returns the current time corresponding to the current date and time. The result is a value of type `xs:time` that contains the time together with timezone information.

### Examples

---

- Calling this function can return a value such as `17:22:33.161Z`:

```
current-time()
```

## fn:data

Return sequence of atomic items.

## Syntax

---

```
fn:data(item* $srcval) => anySimpleType*
```

## Description

---

This function takes a sequence of items as argument and returns a sequence of atomic values. If the sequence is empty, an empty sequence is returned. If the item is an atomic value, it is returned. If the item is a node, it returns the typed value of the node.

## Argument

---

**\$srcval**  
item value

## Example

---

- Get the typed value of name nodes:

```
for $a in input()//name
return ($a, data($a))
```

Note that in general the typed value differs from the string value that is returned by the function [fn:string\(\)](#). This may also apply to nodes that have the type [xs:string](#), if the whitespace facet was used in the schema.



## fn:day-from-date

Return day integer value from date argument.

### Syntax

```
fn:day-from-date(xs:date? $arg) => xs:integer?
```

### Description

This function returns an integer value between 1 and 31, representing the day component of the localized value of the argument value. The localized value is the value in its original timezone or without timezone. If the argument is an empty sequence, the empty sequence is returned.

### Argument

**\$arg**

argument value of type `xs:date`

### Examples

- This function call returns the day component of the current date:

```
day-from-date(current-date())
```

- This function call returns the day component of the specified date value which is 31:

```
day-from-date(xs:date("1999-05-31-05:00"))
```

## fn:day-from-dateTime

Return day integer value from datetime argument.

### Syntax

---

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

### Description

---

This function returns an integer value between 1 and 31, representing the day component of the localized value of the argument value. The localized value is the value in its original timezone or without timezone. If the argument is an empty sequence, the empty sequence is returned.

### Argument

---

#### **\$arg**

argument value of type `xs:dateTime`

### Examples

---

- This function call returns the day component of the current date and time:

```
day-from-dateTime(current-dateTime())
```

- This function call returns the day component of the specified date time value which is 31:

```
day-from-dateTime(xs:dateTime("1999-05-31T13:20:00-05:00"))
```

## fn:deep-equal

Check for items in two arguments that compare equal in corresponding positions.

### Syntax

```
fn:deep-equal(item()* $param1, item()* $param2) => xs:boolean
```

```
fn:deep-equal(item()* $param1, item()* $param2, string $collation) => xs:boolean
```

### Description

This function checks whether two sequences are deep-equal to each other. *Deep equality* means that the sequence must contain items that are pairwise deep-equal. Two items are deep-equal when they are either atomic values that compare equal or when they are nodes of the same kind with the same name whose children are deep-equal. More precisely, the following rules apply:

- If the two sequences are 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 `$param1` is deep-equal to the item at the same position in the sequence `$param2`. Two items `$item1` and `$item2` are deep-equal if one of the following rules holds:
  - If `$item1` and `$item2` are both atomic values, they are deep-equal if and only if `($item1 eq $item2)` is `true`. They are also deep-equal, if both values are `NaN`. If the `eq` operator is not defined for `$item1` and `$item2`, the function returns `false`.
  - If one item is an atomic value and the other is a node, `false` is returned.
  - If both are nodes the following applies: If they are nodes of different kind, the function returns `false`. Otherwise:

Node Kind	<code>\$item1</code> and <code>\$item2</code> are deep-equal, if and only if...
document node	the sequence <code>\$item1/( *   text() )</code> is deep-equal to <code>\$item2/( *   text() )</code> .
element node	<p>all of the following conditions hold:</p> <ol style="list-style-type: none"> <li>1. the two nodes have the same name, that is <code>(node-name(\$item1) eq node-name(\$item2))</code></li> <li>2. the two nodes have the same number of attributes, and for every attribute <code>\$attr1</code> in <code>\$item1/@*</code> there exists an attribute <code>\$attr2</code> in <code>\$item2/@*</code> such that <code>\$attr1</code> and <code>\$attr2</code> are deep-equal. The order of attributes is not significant.</li> </ol>

Node Kind	<code>\$item1</code> and <code>\$item2</code> are deep-equal, if and only if...
	3. One of the following holds: Both nodes have a type annotation that is either a simple type or a complex type with simple content, and the typed value of <code>\$item1</code> is deep-equal to the typed value of <code>\$item2</code> . Or, one or both of the element nodes has a type annotation that is neither a simple type nor a complex type with simple content, and the sequence <code>\$item1/( *   text() )</code> is deep-equal to the sequence <code>\$item2/( *   text() )</code> .
attribute node	the two nodes have the same name, that is <code>(node-name(\$item1) eq node-name(\$item2))</code> , and if the typed value of <code>\$item1</code> is equal to the typed value of <code>\$item2</code> .
processing instruction node	the two nodes have the same name, that is <code>(node-name(\$item1) eq node-name(\$item2))</code> , and if the string value of <code>\$item1</code> is equal to the string value of <code>\$item2</code> .
namespace binding	
text node	their string values are equal.
comment node	

You can also supply a collation argument that is used at all recursion levels where strings are compared.

## Arguments

---

**\$param1**

sequence

**\$param2**

sequence

**\$collation**

valid collation string literal

## Examples

---

Consider the following expression and the subsequent calls of `fn:deep-equal()`:

```
let $at := <attendees>
  <name last='Parker' first='Peter' />
  <name last='Barker' first='Bob' />
  <name last='Parker' first='Peter' />
</attendees>
```

- `deep-equal($at, $at/*)` **returns** false.
- `deep-equal($at/name[1], $at/name[2])` **returns** false.
- `deep-equal($at/name[1], $at/name[3])` **returns** true.
- `deep-equal($at/name[1], 'Peter Parker')` **returns** false.

## fn:distinct-values

Return sequence with distinct values.

### Syntax

---

```
fn:distinct-values(atomicValue* $srcval) => atomicValue
```

```
fn:distinct-values(atomicValue* $srcval, string $collation) => atomicValue
```

### Description

---

This function returns the sequence that contains each value at most once by removing from `$srcval` all but one of a set of equal values. All the values must be of a single type or one of its subtypes; numeric values are promoted to a single common type. Equality must be defined for the type. If the values are string values, then equality is determined according to the collation used. If the values are of type `xs:double` or `xs:float`, the values "0.0" and "-0.0" are considered equal. If values are of type `xs:date` or `xs:time` and have no timezone information, an implicit timezone is provided by the evaluation context.

If the argument is the empty sequence, the function returns the empty sequence.

You can use an optional collation argument which is used when string comparison is required.

### Arguments

---

**\$srcval**

atomic value

**\$collation**

valid collation string literal

### Examples

---

- The following query results only one value of type `xs:time`, since both values are equal:

```
distinct-values((xs:time("10:00:00"), xs:time("11:00:00+01:00")))
```

- The following query results in a type error, since the values of the sequence are not of a single common type:

```
distinct-values((1, "one"))
```

- Get the names of all doctors, each of them appearing only once in the resulting sequence:

```
let    $a := input()/doctor/name/surname  
return distinct-values($a)
```

## fn:ends-with

Check whether a string ends with another string.

### Syntax

---

```
fn:ends-with(string $term, string? $searchString) => boolean
```

```
fn:ends-with(string $term, string? $searchString, string $collation) => boolean
```

### Description

---

This function checks whether the value of `$string` ends with the value of `$searchString` according to the collation used. It returns `true` if it is found and `false` if not. If the value of `$string` or `$searchString` is the empty sequence, the function returns the empty sequence. If the value of `$searchString` is the empty string, then the function returns `"true"`. If the value of `$string` is the empty string and the value of `$searchString` is a non-empty string, the function returns `"false"`.

You can use an optional collation argument.

### Arguments

---

**\$term**

string value

**\$searchString**

string value that is searched for in `$string`

**\$collation**

optional valid collation string literal

### Example

---

- Does "Tamino" end with "ino" ?



```
ends-with("Tamino", "ino")
```

The function returns `true`, since "ino" obviously appears at the end of "Tamino".

## fn:expanded-QName

Return a constructed QName.

### Syntax

---

```
fn:expanded-QName(string $URI, string $localName) => QName
```

### Description

---

This function returns a **QName** with the namespace URI specified in `$URI` and the local name from the value of `$localName`.

### Arguments

---

**\$URI**

namespace URI.

**\$localName**

local name of namespace.

### Examples

---

- Construct the expanded QName `http://company.dot.com/namespaces/corporate:`

```
expanded-QName("http://company.dot.com", "namespaces/corporate")
```

## **fn:false**

Return the boolean value `false`.

### **Syntax**

---

```
fn:false => boolean
```

### **Description**

---

This function returns the boolean value `false`.

## fn:floor

Return largest integer not greater than its argument.

### Syntax

---

```
fn:floor(double? $number) => double?
```

### Description

---

This function returns the largest integer that is not greater than the value of `$number`. If the argument is the empty sequence, it returns the empty sequence. If the argument is a literal, then the literal must be a valid lexical form of its type.

### Argument

---

**\$number**

double-precision number

### Examples

---

- Call function with the numeric value `42.7`:

```
floor(42.7E0)
```

```
floor(xs:float("42.7"))
```

In both cases the double-precision number `4.2E1` is returned. In the first query the argument is a valid literal of the required type, in the second query the required type is achieved by constructing a value of type `xs:float` that is promoted to `xs:double`.

- Call function with the numeric value `-273.15`:

```
floor(-273.15E0)
```

This query returns -2.74E2.

## fn:get-local-name-from-QName

Return local part of QName argument.

### Syntax

---

```
fn:get-local-name-from-QName(QName? $qName) => string?
```

### Description

---

This function returns a string representing the local name of \$qName. If \$qName is the empty sequence, the function returns the empty sequence.

### Argument

---

**\$qName**

value of type xs:QName

### Example

---

- Retrieve local name from QName `http://company.dot.com/namespaces/corporate:`

```
get-local-name-from-QName(expanded-QName("http://company.dot.com", ↵  
"namespaces/corporate"))
```

This query returns the string `namespaces/corporate`.

## fn:get-namespace-from-QName

Return namespace URI of QName argument.

### Syntax

```
fn:get-namespace-from-QName(QName? $qName) => anyURI?
```

### Description

This function returns the namespace URI for `$qName`. If `$qName` is in no namespace, the function returns the empty sequence.

### Argument

**\$qName**

value of type `xs:QName`

### Example

- Retrieve namespace from QName `http://company.dot.com/namespaces/corporate:`

```
get-namespace-from-QName(expanded-QName("http://company.dot.com", ↵  
"namespaces/corporate"))
```

This query returns the string `http://company.dot.com`.

## fn:hours-from-dateTime

Return hours integer value from datetime argument.

### Syntax

---

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

### Description

---

This function returns an integer value between 0 and 23, representing the hours component of the localized value of the argument value. The localized value is the value in its original timezone or without timezone. If the argument is an empty sequence, the empty sequence is returned.

### Argument

---

**\$arg**

argument value of type `xs:dateTime`

### Examples

---

- This function call returns the hours component of the current date and time:

```
hours-from-dateTime(current-dateTime())
```

- This function call returns the hours component of the specified datetime value which is 21:

```
hours-from-dateTime(xs:dateTime("1999-12-31T21:20:00-05:00"))
```



## fn:hours-from-time

Return hours integer value from time argument.

### Syntax

```
fn:hours-from-time(xs:time? $arg) => xs:integer?
```

### Description

This function returns an integer value between 0 and 23, representing the hours component of the localized value of the argument value. The localized value is the value in its original timezone or without timezone. If the argument is an empty sequence, the empty sequence is returned.

### Argument

**\$arg**

argument value of type `xs:time`

### Examples

- This function call returns the hours component of the current time:

```
hours-from-time(current-time())
```

- This function call returns the hours component of the specified time value which is 1:

```
hours-from-time(xs:time("01:23:00+05:00"))
```

## fn:id

Return sequence of element nodes referenced by IDREF values.

## Syntax

---

```
fn:id(xs:string* $arg, node() $node) => element()*
```

## Description

---

This function returns a sequence, in document order with duplicates eliminated, of those elements that are in the same document as `$node` that have an ID value matching one or more of the `xs:IDREFs` in the list of candidate `xs:IDREFs`. The list of `xs:IDREFs` results from parsing the string argument: the string is treated as a sequence of tokens separated by space, and each of the tokens is considered a value of type `xs:IDREF`. If the token is not a lexically valid `xs:IDREF` value, it is ignored. If no `xs:IDREF` value matches the ID value of any element, the function returns the empty sequence. The match is performed strictly using Unicode codepoints, that is, without a collation.

## Argument

---

### **\$arg**

string that can be tokenized into valid values of type `xs:IDREF`

### **\$node**

a node

## fn:idref

Return sequence of element nodes with IDREF values containing at least a specified ID value.

## Syntax

---

```
fn:idref(xs:string* $arg, node() $node) => element()*
```

## Description

---

This function returns a sequence, in document order, of those elements that are in the same document as `$node` that have an IDREF value matching one or more of the `xs:IDs` in the list of candidate `xs:IDs`. The list of `xs:IDs` results from parsing the string argument: the string is treated as a sequence of tokens separated by space, and each of the tokens is considered a value of type `xs:ID`. If the token is not a lexically valid `xs:ID` value, it is ignored. If no `xs:ID` value matches the IDREFS value of any element, the function returns the empty sequence. The match is performed strictly using Unicode codepoints, that is, without a collation.

This function allows reverse navigation from `xs:ID` values or DTD ID values to `xs:IDREF`, `xs:IDREFS` or the DTD types IDREF or IDREFS values.

## Argument

---

### **\$arg**

string that can be tokenized into valid values of type `xs:ID`

### **\$node**

a node

## fn:last

Return number of items in the current sequence.

## Syntax

---

```
fn:last => unsignedInt?
```

## Description

---

This context function returns the number of items in the sequence of items that is currently being processed. The function returns the empty sequence if the context is the empty sequence.

## Example

---

- Retrieve the current German chancellor:

```
let $chancellors := ("Adenauer", "Erhard", "Kiesinger", "Brandt", "Schmidt", ↵  
"Kohl", "Schröder", "Merkel")  
return  
<info>Current German chancellor is: { $chancellors[last()] }.</info>
```

The query returns that string value from `$chancellors` sequence which is the last sequence item.

## fn:local-name

Return local name of node.

## Syntax

```
fn:local-name(node $node) => string
```

## Description

This function returns the local part of the name of `$node`. If `$node` has no name, because it is a document node, a comment node, or a text node, the function returns a string of length zero.

## Argument

**\$node**  
node

## Examples

- Retrieve local name from element nodes:

```
let $a := input()/bib/book[1]
return local-name($a)
```

This query returns the string value "book"

- Retrieve local name from comment nodes:

```
for $a in input()//comment()
return local-name($a)
```

This query returns as many zero-length string values as there are comment nodes in the current collection.

## fn:lower-case

Return lower-cased value of a string.

### Syntax

---

```
fn:lower-case(xs:string $arg) => xs:string
```

### Description

---

This function returns the value of the string argument in lower case by translating each character in its lower case equivalent on its own. If the value of `$arg` is the empty sequence, the string of length zero is returned.

Note that not every character has a lower case equivalent. Also, the case mapping functions are not inverse to each other. This means that **neither** `upper-case(lower-case($arg))` **nor** `lower-case(upper-case($arg))` are guaranteed to yield `$arg`.

### Argument

---

**\$arg**  
string value

### Example

---

- This query returns the string value "abc!d":

```
lower-case("ABc!D")
```

## fn:matches

Matches a string with a pattern.

### Syntax

---

```
fn:matches(string $string, string $pattern) => boolean
```

```
fn:matches(string $string, string $pattern, string $flags) => boolean
```

### Description

---

The function returns true if `$string` matches the regular expression supplied as `$pattern` as influenced by the value of `$flags`, if present; otherwise, it returns false. This function provides pattern-matching functionality which might prove useful for users familiar with this concept. For searching in large data sets, however, it is recommended to rather use Tamino's own text retrieval facilities for better performance.

### Argument

---

**\$sequence**

a string to be matched

**\$pattern**

a pattern

**\$flags**

modifiers for patternmatching

### Example

---

- For each author return the name and a boolean indicating whether it ends with an 'l'. `$` is an anchor character denoting the end of a string.

```
for $author in distinct-values(input()//author/last)
return
<author>
<name>{$author}</name>
<match>{matches($author,'l$')}</match>
</author>
```

The query returns the four different authors wherein `Abiteboul` is marked with `true`. If the pattern is changed to `'l $'` `Abiteboul` does no longer match. It matches again when using the flag string `'x'` as third parameter, since then whitespace characters in the regular expression are removed prior to matching.



## fn:max

Return object with maximum value from item sequence.

## Syntax

```
fn:max(sequence $sequence) => anySimpleType?
```

```
fn:max(sequence $sequence, string $collation) => anySimpleType?
```

## Description

This aggregate function returns the maximum value of all values in a sequence that are of a single type or one of its subtypes, for which the `gt` operator is defined. For numeric values type promotion is used to coerce all values to a single common type. If the argument is the empty sequence, the function returns the empty sequence.

## Argument

### **\$sequence**

a sequence of items

### **\$collation**

optional valid collation string literal

### **\$flags**

modifiers for patternmatching

## Example

- Get maximum value from sequence with numeric values:

```
let $a := (17, 23, 11)
return max($a)
```

## fn:min

Return object with minimum value from item sequence.

## Syntax

---

```
fn:min(sequence $sequence) => anySimpleType?
```

```
fn:min(sequence $sequence, string $collation) => anySimpleType?
```

## Description

---

This aggregate function returns the minimum value of all values in a sequence that are of a single type or one of its subtypes, for which the `<` operator is defined. For numeric values type promotion is used to coerce all values to a single common type. If the argument is the empty sequence, the function returns the empty sequence.

## Argument

---

### **\$sequence**

a sequence of items

### **\$collation**

optional valid collation string literal

## Examples

---

- Get minimum value from sequence with numeric values:

```
let $a := (17, 23, 11)
return min($a)
```

## fn:minutes-from-dateTime

Return minutes integer value from datetime argument.

### Syntax

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

### Description

This function returns an integer value between 0 and 59, representing the minutes component of the localized value of the argument value. The localized value is the value in its original timezone or without timezone. If the argument is an empty sequence, the empty sequence is returned.

### Argument

**\$arg**

argument value of type `xs:dateTime`

### Examples

- This function call returns the minutes component of the current date and time:

```
minutes-from-dateTime(current-dateTime())
```

- This function call returns the minutes component of the specified datetime value which is 20:

```
minutes-from-dateTime(xs:dateTime("1999-12-31T21:20:00-05:00"))
```

## fn:minutes-from-time

Return minutes integer value from time argument.

### Syntax

---

```
fn:minutes-from-time(xs:time? $arg) => xs:integer?
```

### Description

---

This function returns an integer value between 0 and 59, representing the minutes component of the localized value of the argument value. The localized value is the value in its original timezone or without timezone. If the argument is an empty sequence, the empty sequence is returned.

### Argument

---

**\$arg**

argument value of type `xs:time`

### Examples

---

- This function call returns the minutes component of the current time:

```
minutes-from-time(current-time())
```

- This function call returns the minutes component of the specified time value which is 23:

```
minutes-from-time(xs:time("01:23:00+05:00"))
```

## fn:month-from-date

Return month integer value from date argument.

### Syntax

```
fn:month-from-date(xs:date? $arg) => xs:integer?
```

### Description

This function returns an integer value between 1 and 12, representing the month component of the localized value of the argument value. The localized value is the value in its original timezone or without timezone. If the argument is an empty sequence, the empty sequence is returned.

### Argument

**\$arg**

argument value of type `xs:date`

### Examples

- This function call returns the month component of the current date:

```
month-from-date(current-date())
```

- This function call returns the month component of the specified date value which is 5:

```
month-from-date(xs:date("1999-05-31-05:00"))
```

## fn:month-from-dateTime

Return month integer value from datetime argument.

### Syntax

---

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

### Description

---

This function returns an integer value between 1 and 12, representing the month component of the localized value of the argument value. The localized value is the value in its original timezone or without timezone. If the argument is an empty sequence, the empty sequence is returned.

### Argument

---

**\$arg**

argument value of type `xs:dateTime`

### Examples

---

- This function call returns the month component of the current date and time:

```
month-from-dateTime(current-dateTime())
```

- This function call returns the month component of the specified date time value which is 5:

```
month-from-dateTime(xs:dateTime("1999-05-31T13:20:00-05:00"))
```

## fn:namespace-uri

Return namespace URI from node.

### Syntax

```
fn:namespace-uri(node $node) => anyURI
```

```
fn:namespace-uri => string
```

### Description

This function returns the namespace URI of a node if it is qualified by a namespace. If *\$node* is present, it returns its namespace URI, otherwise it returns the namespace URI of the context node. If the respective node is not qualified, the function returns the empty string.

### Argument

**\$node**  
node

### Examples

- Retrieve namespace URI from previously declared namespace:

```
declare namespace dotcom = "http://company.dot.com/namespaces/corporate"  
namespace-uri(<dotcom:bookshelf/>)
```

The query returns the namespace URI *http://company.dot.com/namespaces/corporate*.

- Retrieve namespace URI from unqualified namespace:

```
namespace-uri(<unqualified/>)
```

This query returns the empty string.

## fn:node-name

Return expanded QName of a node.

### Syntax

---

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

### Description

---

This function returns an expanded QName for those kinds of nodes 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.

### Argument

---

**\$arg**  
any kind of node



## fn:normalize-space

Return its argument with normalized whitespace.

### Syntax

```
fn:normalize-space(string? $string) => string?
```

### Description

This function returns the argument with normalized whitespace. Normalizing whitespace includes stripping leading and trailing whitespace and replace more than one whitespace character with a single whitespace character.

### Argument

**\$string**  
string value

### Example

- Return the normalized form of a string with lots of spaces:

```
normalize-space("  lots of spaces everywhere  ...  ")
```

This returns the string "lots of spaces everywhere ...".

## fn:not

Invert boolean value of its argument.

## Syntax

---

```
fn:not(sequence $sequence) => boolean
```

## Description

---

This function returns `true`, if the effective boolean value of `$sequence` is `false`, and it returns `false`, if the effective boolean value of `$sequence` is `true`.

## Argument

---

### **\$sequence**

Description missing.

## Examples

---

- Negate an empty sequence:

```
not(())
```

The result is `true`, since the effective boolean value of the empty sequence is `false`.

- Select all patients that do not have some form of angina:

```
for $a in input()/patient
let $b := $a/submitted/diagnosis
where not(starts-with($b, "Angina"))
return $a/name
```

## fn:position

Return position of context item in current sequence.

## Syntax

```
fn:position => unsignedInt?
```

## Description

This context function returns the position of the context item within the sequence of items that is currently being processed. The function returns the empty sequence if the context is the empty sequence.

## Example

- Retrieve the first even natural numbers:

```
let $numbers := ("1", "2", "3", "4", "5", "6", "7", "8", "9", "10" )  
return $numbers[position() mod 2 = 0]
```

The query returns all those sequence items whose position modulo 2 equals 0, i.e., which is an even number.

## fn:put

Inserts a new instance into a database.

## Syntax

---

```
fn:put(node $node, string $uri)
```

## Description

---

The function does not return a result.

Updating functions, which include `fn:put`, cannot be called from an XQuery expression. This function can only be called within an XQuery update expression.

If the `$node` argument holds an XML document node, a corresponding document is stored in the Tamino database. If the `$node` argument holds an element, it is internally wrapped by a document node.

The `fn:put` function is restricted to store documents in the database instance that is processing the update expression. Because of this constraint, the argument must be a relative URI. The relative URI is interpreted in the same way as the corresponding part in the URI of a `_process` command; this means that:

- the collection is mandatory;
- in the case of an XML document, the doctype part may be empty; otherwise, the document is stored with the specified doctype;
- the docname part is optional.

The doctype part of the URI must be indicated by a slash. The docname part of the URI, if present, must be indicated by a slash. The following example URI stores into `ino:etc` with a docname of `"test"`: `ino:etc//test`.

## Parameters

---

### **\$node**

A node that holds the instance to be inserted.

### **\$uri**

The URI of the location where the instance should be stored.

## Examples

---

- An update expression that directly calls the `fn:put` function. It inserts a document that holds an author list into the *bib* collection.

```
update
fn:put(document{<author-list>{collection("bib")/bib/book/author}</author-list>},"bib")
```

- Using a FLWU expression improves the readability:

```
update
let $node := ←
document{<author-list>{collection("bib")/bib/book/author}</author-list>}
do fn:put($node,"bib")
```

## fn:replace

Replaces bits of a string using patternmatching.

### Syntax

---

```
fn:replace(string $input, string $pattern, string $replacement) => string
```

```
fn:matches(string $input, string $pattern, string $replacement, string $flags) ←  
=> string
```

### Description

---

The function returns the input string wherein all occurrences of `$pattern` are replaced by `$replacement` influenced by the value of `$flags`, if present.

### Argument

---

**\$input**

a string to be modified

**\$pattern**

a pattern

**\$replacement**

a string to be inserted

**\$flags**

modifiers for patternmatching

### Example

---

- For each string replace "x" with "y".

```
for $value in ("axa","bxb","cxc")  
return replace($value,"x","y")
```

The query returns:

```
<xq:result>  
  <xq:value>aya</xq:value>  
  <xq:value>byb</xq:value>  
  <xq:value>cyc</xq:value>  
</xq:result>
```

## fn:reverse

Reverses the order of items in a sequence.

### Syntax

---

```
fn:reverse(sequence $sequence) => sequence
```

### Description

---

This function returns the input sequence in reversed order.

### Argument

---

**\$sequence**  
sequence

### Example

---

- Retrieve the last name of every author and reverse the obtained sequence:

```
declare namespace ino="http://namespaces.softwareag.com/tamino/response2"
let $authors := input()//author/last
return ($authors,<delimiter/>,reverse($authors))
```

The query returns all authors and, separated by a constructed delimiter-node, the authors in reversed order.



## fn:root

Return root of tree with argument node.

## Syntax

```
fn:root(node $node) => node
```

## Description

This function returns the root of the tree to which `$node` belongs. For persistent nodes this is always a document node, for transient nodes this is not necessarily the case.

## Argument

**\$node**  
node

## Example

- Retrieve the modules stored in the current collection as non-XML data:

```
declare namespace ino="http://namespaces.softwareag.com/tamino/response2"
root(
  for $a in collection("ino:source")/ino:module
  where $a/@ino:targetNamespace="http://www.examples.com/tree"
  return $a
)
```

The query returns all those modules as document nodes that are in the namespace *http://www.examples.com/tree*.

## fn:round

Return number closest to its argument.

### Syntax

---

```
fn:round(double? $number) => double?
```

### Description

---

This function returns the number that is closest to `$number`. More precisely, `fn:round(x)` returns the same result as `fn:floor(x + 0.5)`. If the argument is the empty sequence, it returns the empty sequence. If the argument is a literal, then the literal must be a valid lexical form of its type.

### Argument

---

**\$number**

double-precision number

### Examples

---

- Call function with the numeric value `41.5`:

```
round(41.5E0)
```

```
round(xs:double("41.7"))
```

In both cases the double-precision number `4.2E1` is returned. In the first query the argument is a valid literal of the required type, in the second query the required type is constructed using the constructor function `xs:double`.

- Call function with the numeric value `-273.15`:

```
round(-273.15E0)
```

This query returns -2.73E2.

## fn:seconds-from-dateTime

Return seconds integer value from datetime argument.

### Syntax

---

```
fn:seconds-from-dateTime(xs:dateTime? $arg) => xs:integer?
```

### Description

---

This function returns an integer value between 0 and 60.999, representing the seconds component of the localized value of the argument value. The localized value is the value in its original timezone or without timezone. If the argument is an empty sequence, the empty sequence is returned.

Please note that the value can be beyond 60 in order to account for leap seconds.

### Argument

---

**\$arg**

argument value of type `xs:dateTime`

### Examples

---

- This function call returns the seconds component of the current date and time:

```
seconds-from-dateTime(current-dateTime())
```

- This function call returns the seconds component of the specified datetime value which is 0:

```
seconds-from-dateTime(xs:dateTime("1999-12-31T21:20:00-05:00"))
```

## fn:seconds-from-time

Return seconds integer value from time argument.

### Syntax

```
fn:seconds-from-time(xs:time? $arg) => xs:integer?
```

### Description

This function returns an integer value between 0 and 60.999, representing the seconds component of the localized value of the argument value. The localized value is the value in its original timezone or without timezone. If the argument is an empty sequence, the empty sequence is returned.

Please note that the value can be beyond 60 in order to account for leap seconds.

### Argument

**\$arg**

argument value of type `xs:time`

### Examples

- This function call returns the seconds component of the current time:

```
seconds-from-time(current-time())
```

- This function call returns the seconds component of the specified time value which is 0:

```
seconds-from-time(xs:time("01:23:00+05:00"))
```

## fn:starts-with

Check whether string 1 starts with string 2

### Syntax

---

```
fn:starts-with(string $string, string $startString) => boolean
```

```
fn:starts-with(string $string, string $startString, string $collation) => boolean
```

### Description

---

This function checks whether or not the value of `$string` starts with the value of `$startString`. If the value of `$startString` is a string of length zero, the function returns `true`. If the value of `$string` is a string of length zero, but the value of `$startString` is not, the function returns `false`. If the value of either string is the empty sequence, the function returns the empty sequence.

### Arguments

---

**\$string**

string value

**\$startString**

string value to be searched for in `$string`

**\$collation**

optional valid collation string literal

### Example

---

- Select all patients that have some form of angina:

```
for $a in input()/patient
let $b := $a/submitted/diagnosis
where starts-with($b, "Angina")
return $a/name
```

## fn:string

Return argument value as a string.

## Syntax

```
fn:string(item $value) => string
```

## Description

This accessor function returns the value of `$value` represented as a string. If `$value` is a non-empty sequence, a type exception is raised. If `$value` is an empty sequence or an empty string, a string of length zero is returned as result. If `$value` is a node, the string value of the node is returned. If `$value` is an atomic value, it is converted to a string and returned.

Note that the value returned by this function, when applied to a node, may differ from the typed value, even if the node is specified in the schema as being of type **xs:string**. To access the typed value, use the function **fn:data**.

## Argument

### `$value`

one of an atomic value, a string or the string value of a node

## Examples

- Get a sequence of numeric values as a string value:

```
string((1,2,3))
```

This query results in a type exception.

```
string("(1,2,3)")
```

This query returns the string value `"(1,2,3)"`.

- Get a namespace as a string value:

```
string(xs:anyURI("http://no%20space%20lost"))
```

If URIs contain special characters, they will not be escaped.



## fn:string-join

Return concatenation of string sequence.

### Syntax

```
fn:string-join(string* $stringSequence, string $separator) => string?
```

### Description

This function returns a possibly empty string that is created by concatenating the members of `$stringSequence` using the value of `$separator`. If the value of `$separator` is a string of length zero, the members of `$stringSequence` are concatenated without a separator.

### Arguments

**`$stringSequence`**  
string value

**`$separator`**  
string value

### Examples

- Some variants of simple string concatenation using different separators:

```
string-join(('do', 're', 'mi', 'fa', 'so', 'la', 'si'), "")
```

```
string-join(("veni", "vidi", "vici"), ", ")
```

```
string-join(("line 1", "line 2", "line3"), "&#xA;")
```

The first function call returns the string "doremifasolasi", the second returns "veni, vidi, vici", and the third call returns:

```
line 1  
line 2  
line 3
```

- For all doctors with pagers whose numbers start with a "3" change the pager number by prepending "11-":

```
update for $a in input()//doctor  
  let $b := $a/@pager  
  where starts-with($b, "3")  
  do replace $b  
  with attribute pager { string-join(("11", $b), "-") }
```

## fn:string-length

Return length of string value.

### Syntax

```
fn:string-length(string? $string) => integer?
```

### Description

This function returns the length of the value of `$string` as integer value. If the value of `$string` is the empty sequence, the empty sequence is returned.

### Argument

**\$string**  
string value

### Example

```
■ string-length("Donaudampfschiffahrtsskapitänsabzeichen")
```

```
■ string-length(normalize-space("  "))
```

The first function call returns 38, the second zero, since the string value is a zero-length string after normalizing whitespace.

## fn:subsequence

Return subsequence of a sequence.

### Syntax

---

```
fn:subsequence(sequence $sequence, double $start) => sequence
```

```
fn:subsequence(sequence $sequence, double $start, double $length) => sequence
```

### Description

---

This function returns a subsequence of `$sequence`, which begins at `$start` and contains the next `$length` entries, if a third parameter is provided.

### Argument

---

**\$sequence**

sequence

**\$start**

double value: the position at which the subsequence starts

**\$length**

double value: the length of the subsequence

### Examples

---

- Return all entries starting from the 5<sup>th</sup>:

```
let $seq := (1,2,3,4,5,6,7)
return subsequence($seq,5)
```

This returns (5,6,7).

- Return three entries starting from the 2<sup>nd</sup>:

```
let $seq := (1,2,3,4,5,6,7)
return subsequence($seq,2,3)
```

This returns (2,3,4).

## fn:substring

Return substring of a string value.

### Syntax

---

```
fn:substring(string $string, double $position) => string?
```

```
fn:substring(string $string, double $position, double $length) => string?
```

### Description

---

This function returns that part of `$string`, which begins at `$position` and is the number of characters indicated by `length`. More precisely, the function returns the characters in `$string`, whose position `$p` satisfies:

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

The following rules hold:

- If `$length` is not specified or if `$length` is greater than the number of characters in the value of `$string` following `$position`, the substring includes characters to the end of `$string`.
- If `$position` is zero or negative, the substring includes characters from the beginning of the `$string`.
- It is an error if `$string` is the empty sequence.

### Argument

---

**`$string`**

string value

**`$position`**

double value: the position at which the search takes place

**`$length`**

double value: the length of the substring

### Examples

---

- Both queries below return "XML":

```
substring("Tamino XML Server", 7, 4)
substring("Tamino XML Server", 6.5, 4.4)
```

Like the first query, the second one looks for four characters beginning at position 7.

- Return the second half of the specified string:

```
let    $text := "Tamino XML Server"
return substring($text, string-length($text) div 2)
```

## fn:substring-after

Return substring of a string value following another string.

### Syntax

---

```
fn:substring-after(string $string, string? $searchString) => string?
```

```
fn:substring-after(string $string, string? $searchString, string $collation) ↵  
=> string?
```

### Description

---

This function returns that part of `$string`, which follows the string value `searchString`.

If the value of `$string` or `$searchString` is the empty sequence, the function returns the empty sequence. If the value of `$searchString` is the empty string, then the function returns the value of `$string`. If the value of `$string` does not contain a string that is equal to the value of `$searchString`, then the function returns the empty string

### Argument

---

**`$string`**

string value

**`$searchString`**

string value that is searched in `$string`

**`$collation`**

optional valid collation string literal

### Example

---

This query returns the string value "Server":



```
substring-after("Tamino XML Server", "XML")
```

## fn:substring-before

Return substring of a string value preceding another string.

### Syntax

---

```
fn:substring-before(string $string, string? $searchString) => string?
```

```
fn:substring-before(string $string, string? $searchString, string $collation) ↵  
=> string?
```

### Description

---

This function returns that part of `$string`, which precedes the string value `searchString`.

If the value of `$string` or `$searchString` is the empty sequence, the function returns the empty sequence. If the value of `$searchString` is the empty string, then the function returns the value of `$string`. If the value of `$string` does not contain a string that is equal to the value of `$searchString`, then the function returns the empty string

### Argument

---

**\$string**

string value

**\$searchString**

string value that is searched in `$string`

**\$collation**

optional valid collation string literal

### Example

---

This query returns the string value "Tamino ":

```
substring-before("Tamino XML Server", "XML")
```

## fn:sum

Return sum of a sequence of numbers.

### Syntax

---

```
fn:sum(sequence $sequence) => double
```

### Description

---

This aggregate function returns the sum of all numeric values in a sequence. If the argument is an empty sequence, the function returns the numerical value 0.

### Argument

---

**\$sequence**  
sequence of numerical values

### Examples

---

- This query returns as sum of the sequence values the numerical value "4.2E1" of type `xs:double`:

```
let    $a := (17, 23, 11, -10)
return sum($a)
```

## fn:tokenize

Tokenizes a string according to a pattern.

### Syntax

```
fn:tokenize(string $string, string $pattern) => string*
```

```
fn:tokenize(string $string, string $pattern, string $flags) => string*
```

### Description

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

### Argument

**\$string**

a string to be tokenized

**\$pattern**

a separator pattern

**\$flags**

modifiers for patternmatching

### Example

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

The query returns ("Some unparsed", "HTML", "text") the pattern `"\s*"` denoting any number of whitespace characters.

## **fn:true**

Return the boolean value `true`.

### **Syntax**

---

```
fn:true => boolean
```

### **Description**

---

This function returns the boolean value `true`.

## fn:upper-case

Return upper-cased value of a string.

### Syntax

```
fn:upper-case(xs:string $arg) => xs:string
```

### Description

This function returns the value of the string argument in upper case by translating each character in its upper-case equivalent on its own. If the value of `$arg` is the empty sequence, the string of length zero is returned.

Note that not every character has an upper case equivalent. Also, the case mapping functions are not inverse to each other. This means that neither `upper-case(lower-case($arg))` nor `lower-case(upper-case($arg))` are guaranteed to yield `$arg`.

### Argument

**\$arg**  
string value

### Example

- This query returns the string value "ABC!D":

```
lower-case("aBc!D")
```

## fn:year-from-date

Return year integer value from date argument.

### Syntax

---

```
fn:year-from-date(xs:date? $arg) => xs:integer?
```

### Description

---

This function returns an integer value between 1 and 12, representing the year component of the localized value of the argument value. The localized value is the value in its original timezone or without timezone. The value may be negative. If the argument is an empty sequence, the empty sequence is returned.

### Argument

---

**\$arg**

argument value of type `xs:date`

### Examples

---

- This function call returns the year component of the current date:

```
year-from-date(current-date())
```

- This function call returns the year component of the specified date value which is 1999:

```
year-from-date(xs:date("1999-05-31-05:00"))
```



## fn:year-from-dateTime

Return year integer value from datetime argument.

### Syntax

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

### Description

This function returns an integer value between 1 and 12, representing the year component of the localized value of the argument value. The localized value is the value in its original timezone or without timezone. The value may be negative. If the argument is an empty sequence, the empty sequence is returned.

### Argument

**\$arg**

argument value of type `xs:dateTime`

### Examples

- This function call returns the year component of the current date and time:

```
year-from-dateTime(current-dateTime())
```

- This function call returns the year component of the specified date time value which is 1999:

```
year-from-dateTime(xs:dateTime("1999-05-31T13:20:00-05:00"))
```

## ft:proximity-contains

Search for word tokens within some distance.

### Syntax

---

```
ft:proximity-contains(node $node, string $searchString, integer $distance, boolean $ordered) ←  
=> boolean
```

### Description

---

This text retrieval function searches a node for a sequence of one or more word tokens (passed as a string) within a specified proximity distance and in a specified order. If the argument `$node` evaluates to the empty sequence, `false` is returned.

The `$distance` argument determines how far apart the matched word tokens in the string value of the node may be. The distance is evaluated as the maximum number of unmatched tokens between the first matched word token and the last matched word token in `$searchString`. The function returns `true`, if `$distance` is larger than this computed distance. For example, a value of "1" means they must follow immediately after one another, a value of 2 allows a gap of one word in between etc.

With `ft:proximity-contains` you can perform search operations including the use of a wildcard character. The section *Using Wildcard Characters* in the *XQuery 4 User Guide* explains this in detail.

There are no defaults defined, so you need to invoke it with all arguments. This function is bound to the namespace `http://www.w3.org/2002/04/xquery-operators-text` and you need to declare that namespace in the [query prolog](#).



**Note:** This function is deprecated and will be removed in future versions of Tamino. You should use one of the functions `tf:containsText`, `tf:containsAdjacentText` or `tf:containsNearText` instead. See the examples for details.

### Arguments

---

#### **\$node**

node to be searched

#### **\$searchString**

string containing a sequence of one or more words to be searched for

#### **\$distance**

integer value denoting proximity distance

**\$ordered**

if true, the order of word tokens in `$searchString` is taken into account.

**Examples**

In the patient sample data, there is a `remarks` element for the patient Bloggs that reads: "Patient is responding to treatment. Dr. Shamir."

- Retrieve all patients who are responding to current treatment:

```
declare namespace ft="http://www.w3.org/2002/04/xquery-operators-text"
for $a in input()/patient
where ft:proximity-contains($a/remarks, "to treatment", 1, true())
return $a/name
```

This query returns all the names of all patients for which `ft:proximity-contains` returns true, since the words "to" and "treatment" immediately follow after one another in that order. You can rewrite queries of this kind with `tf:containsAdjacentText`. Note that you have to specify each of the search words as a separate argument:

```
for $a in input()/patient
where tf:containsAdjacentText($a/remarks, 1, "to", "treatment")
return $a/name
```

- Retrieve all patients who are responding to current treatment:

```
declare namespace ft="http://www.w3.org/2002/04/xquery-operators-text"
for $a in input()/patient
where ft:proximity-contains($a/remarks, "treatment responding", 2, false())
return $a/name
```

This query returns all the names of all patients for which `ft:proximity-contains` returns true. In the text contents of the `remarks` node, the word tokens "treatment" and "responding" may have at most one word token in between and appear in either order. You can rewrite queries of this kind with `tf:containsNearText`. Note that you have to specify each of the search words as a separate argument:

```
declare namespace ft="http://www.w3.org/2002/04/xquery-operators-text"
for $a in input()/patient
where tf:containsNearText($a/remarks, 2, "treatment", "to")
return $a/name
```

- Check for each patient if there is the word "treatment" in the remarks.

```
declare namespace ft="http://www.w3.org/2002/04/xquery-operators-text"
for $a in input()/patient
return ($a/name, ft:proximity-contains($a/remarks, "treatment", 0, true()))
```

This form of `ft:proximity-contains` degenerates to a simple text search: a distance of 0 allows no delimiters within the `$searchString` restricting it to a single word token. The order has no effect here. You should use `tf:containsText` instead:

```
for $a in input()/patient
return ($a/name, tf:containsText($a/remarks, "treatment"))
```

## ft:text-contains

Search for word tokens in a search string.

### Syntax

```
ft:text-contains(node $node, string $searchString) => boolean
```

### Description

This text retrieval function searches a node for a sequence of one or more word tokens (passed as a string). It returns `true` if all word tokens appear exactly in the order as specified in `$searchString`.

With `ft:text-contains` you can perform search operations including the use of a wildcard character. The section *Pattern Matching* in the *XQuery 4 User Guide* explains this in detail.

This function is bound to the namespace <http://www.w3.org/2002/04/xquery-operators-text> and you need to declare that namespace in the [query prolog](#).



**Note:** This function is deprecated and will be removed in future versions of Tamino. You should use the function `tf:containsText` instead. See the examples for details.

### Arguments

#### **\$node**

node in which the search takes place

#### **\$searchString**

string containing a sequence of one or more words to be searched

### Example

- Retrieve all patients who are responding to current treatment:

```
declare namespace ft="http://www.w3.org/2002/04/xquery-operators-text"
for $a in input()/patient
where ft:text-contains($a/remarks, "responding to treatment")
return $a/name
```

You can rewrite this query using `tf:containsText`:

```
for $a in input()/patient
where tf:containsText($a/remarks, "responding to treatment")
return $a/name
```

## tdf:getProperties

Get all WebDAV properties of a document.

### Syntax

```
tdf:getProperties(documentroot $resource) => element
```

### Description

The function `tdf:getProperties` is specific to Tamino and its WebDAV functionality. It retrieves all properties of a document stored as WebDAV resource in Tamino. This corresponds to the WebDAV request `PROPFIND`.

### Argument

#### **\$resource**

WebDAV resource which is a document root in Tamino

### Examples

- Show all properties for all documents in the collection `ino:dav`

```
declare namespace ino="http://namespaces.softwareag.com/tamino/response2"
for $x in collection("ino:dav")
return tdf:getProperties($x)
```

## tdf:getProperty

Get single WebDAV property of a document.

### Syntax

---

```
tdf:getProperty(documentroot $resource, xs:string $uri, xs:string $localName) ←  
=> xs:string
```

### Description

---

The function `tdf:getProperty` is specific to Tamino and its WebDAV functionality. It retrieves all properties of a document stored as WebDAV resource in Tamino. This corresponds to the WebDAV request PROPFIND.

### Argument

---

**\$resource**

WebDAV resource which is a document root in Tamino

**\$uri**

namespace URI of the WebDAV property

**\$localName**

local name of the WebDAV property

### Examples

---

- Show all documents in the collection `ino:dav` that use the content type "text/xml". The content type is stored in the WebDAV property `getcontenttype`.

```
declare namespace ino="http://namespaces.softwareag.com/tamino/response2"  
for   $x in collection("ino:dav")  
where tdf:getProperty($x, "DAV:", "getcontenttype") = "text/xml"  
return $x
```



## tdf:isDescendantOf

Check for WebDAV resource on given path.

### Syntax

---

```
tdf:isDescendantOf(documentroot $resource, xs:string $path, xs:string $depth) => bool
```

### Description

---

The function `tdf:isDescendantOf` is specific to Tamino and its WebDAV functionality. It checks whether the WebDAV resource is found on the specified path given the lookup depth. If the resource is found, `true` is returned, otherwise `false`.

### Argument

---

**\$resource**

WebDAV resource which is a document root in Tamino

**\$path**

relative WebDAV path in Tamino

**\$depth**

restricts depth of resource lookup; valid values are:

Value	Meaning
-------	---------

"0"	resource itself
-----	-----------------

"1"	direct descendants
-----	--------------------

"infinity"	all descendants (default value)
------------	---------------------------------

### Examples

---

- Return boolean values for all resources in collection `ino:dav` that are on the relative path `my/path`:

```
declare namespace ino="http://namespaces.softwareag.com/tamino/response2"
for   $x in collection("ino:dav")
return tdf:isDescendantOf($x, "/ino:dav/ino:dav/my/path", "1")
```

## **tdf:mkcol**

Create a new collection resource

### **Syntax**

---

```
tdf:mkcol(string $uri)
```

### **Description**

---

This function has been added in order to complete the changes needed in Tamino's update processor and XQuery implementations.

### **Argument**

---

#### **\$uri**

The location where the new collection resource should be created.

## tdf:resource

Get sequence of WebDAV resources.

### Syntax

---

```
tdf:resource(xs:string $path, xs:string $depth) => sequence
```

### Description

---

The function `tdf:resource` is specific to Tamino and its WebDAV functionality. It looks for WebDAV resources on the specified path given the lookup depth and returns a sequence of matching documents.

### Arguments

---

#### **\$path**

relative WebDAV path in Tamino

#### **\$depth**

restricts depth of resource lookup; valid values are:

Value	Meaning
-------	---------

"0"	resource itself
-----	-----------------

"1"	direct descendants
-----	--------------------

"infinity"	all descendants (default value)
------------	---------------------------------

### Examples

---

- Return the Tamino IDs of all direct descendant resources of `my/path`:

```
declare namespace ino="http://namespaces.softwareag.com/tamino/response2"
for   $x in tdf:resource("/ino:dav/ino:dav/my/path", "1")
return tf:getInoId($x)
```

## tdf:setProperty

Set single WebDAV property of a document.

### Syntax

---

```
tdf:setProperty(node $node, string? $uri, string $localName, item* $propValue)
```

### Description

---

The function `tdf:setProperty` assigns a WebDAV property to an existing document. It accepts a property value of arbitrary complexity. If the empty sequence is passed to the function as the `$propValue` parameter, the specified property is deleted (if it is currently defined for the document). The function is in the namespace `http://namespaces.softwareag.com/tamino/DaoFunction`.

### Arguments

---

**\$node**

the document node

**\$uri**

together with `$localName`, this parameter specifies the QName of the property

**\$localName**

together with `$uri`, this parameter specifies the QName of the property

**\$propValue**

the value of the property

## tf:broaderTerm

Search for immediate superordinate term.

## Syntax

---

```
tf:broaderTerm(string $termName, string? $thesaurus) => unspecified
```

## Description

---

The function `tf:broaderTerm` is specific to Tamino. It takes a string as argument which is the name of a thesaurus entry. You can supply a further optional string value that specifies the name of a thesaurus. In the scope of text retrieval functions this function returns the immediate superordinate term of `$termName`. If there is no superordinate term defined, only `$termName` is returned.

This function can only be used within the scope of the following functions:

```
tf:containsAdjacentText  
tf:containsNearText  
tf:containsText  
tf:createAdjacentTextReference  
tf:createNearTextReference  
tf:createTextReference
```

## Arguments

---

### **\$termName**

name of a thesaurus entry that is defined in the `termName` element

### **\$thesaurus**

name of a thesaurus

## Examples

---

- Return superordinate term of "dog":

```
declare namespace ino="http://namespaces.softwareag.com/tamino/response2"
for   $p in input()/ino:term
where tf:containsText($p/ino:termName, tf:broaderTerm("dog"))
return $p/ino:termName
```

## tf:broaderTerms

Search for all superordinate terms.

### Syntax

---

```
tf:broaderTerms(string $termName, string? $thesaurus) => unspecified
```

### Description

---

The function `tf:broaderTerms` is specific to Tamino. It takes a string as argument which is the name of a thesaurus entry. You can supply a further optional string value that specifies the name of a thesaurus. In the scope of text retrieval functions this function returns all superordinate terms of `$termName`. If there is no superordinate term defined, only `$termName` is returned.

This function can only be used within the scope of the following functions:

```
tf:containsAdjacentText  
tf:containsNearText  
tf:containsText  
tf:createAdjacentTextReference  
tf:createNearTextReference  
tf:createTextReference
```

### Arguments

---

**\$termName**

name of a thesaurus entry that is defined in the `termName` element

**\$thesaurus**

name of a thesaurus

### Examples

---

- Return all superordinate terms of "dog":



```
declare namespace ino="http://namespaces.softwareag.com/tamino/response2"
for   $p in input()/ino:term
where tf:containsText($p/ino:termName, tf:broaderTerms("dog"))
return $p/ino:termName
```

## tf:containsAdjacentText

Search word tokens in order within some distance.

### Syntax

---

```
tf:containsAdjacentText(node $input, integer $distance, token+ $word) => boolean
```

### Description

---

The function `tf:containsAdjacentText` is specific to Tamino. It searches a node for a sequence of one or more word tokens within a specified proximity distance and in token order. If the argument `$node` evaluates to the empty sequence, `false` is returned.

The `$distance` argument determines how far apart the matched word tokens in the node may be. The distance is evaluated as the maximum number of unmatched tokens between the first matched word token and the last matched word token. The function returns `true`, if `$distance` is larger than this computed distance. For example, a value of "1" means they must follow immediately after one another, a value of 2 allows a gap of one word in between etc.

With `tf:containsAdjacentText` you can perform search operations including the use of a wildcard character. The section *Pattern Matching* in the *XQuery 4 User Guide* explains this in detail.

### Arguments

---

**\$input**

node to be searched

**\$distance**

integer value denoting proximity distance

**\$word**

a word token

### Examples

---

- Retrieve all patients who are responding to current treatment:

```
for $a in input()/patient
where tf:containsAdjacentText($a/remarks, 3, "patient", "to", "treatment")
return $a/name
```

This query returns all the names of all patients for which `tf:containsAdjacentText` returns true. This is the case for all `remarks` nodes in which the word tokens "patient", "to" and "treatment" follow after one another in a distance of a most three word tokens, i.e., there may be no more than two word tokens in-between. Given the sample data, Bloggs is the only patient.

Since `tf:containsAdjacentText` takes the order of the word tokens into account, swapping the tokens would yield an empty result set:

```
for $a in input()/patient
where tf:containsAdjacentText($a/remarks, 3, "treatment", "patient", "to")
return $a/name
```

- Retrieve all remarks in which the word "treatment" occurs:

```
for $a in input()/patient
where tf:containsAdjacentText($a/remarks, 1000, "treatment")
return $a/name
```

This is the trivial case of searching only one word token. If the word token is present in the node, the result is true regardless of the specified distance. You can express this equivalently using the function `containsText`:

```
for $a in input()/patient
where tf:containsText($a/remarks, "treatment")
return $a/name
```

## tf:containsNearText

Search word tokens within some distance, but without order.

### Syntax

---

```
tf:containsNearText(node $input, integer $distance, token+ $word) => boolean
```

### Description

---

The function `tf:containsNearText` is specific to Tamino. It searches a node for a sequence of one or more word tokens within a specified proximity distance and in token order. If the argument `$node` evaluates to the empty sequence, `false` is returned.

The `$distance` argument determines how far apart the matched word tokens in the node may be. The distance is evaluated as the maximum number of unmatched tokens between the first matched word token and the last matched word token. The function returns `true`, if `$distance` is larger than this computed distance. For example, a value of "1" means they must follow immediately after one another, a value of 2 allows a gap of one word in between etc.

With `tf:containsNearText` you can perform search operations including the use of a wildcard character. The section *Pattern Matching* in the *XQuery 4 User Guide* explains this in detail.

### Arguments

---

**\$input**

node to be searched

**\$distance**

integer value denoting proximity distance

**\$word**

a word token

### Example

---

- Retrieve all patients who are responding to current treatment:

```
for $a in input()/patient
where tf:containsNearText($a/remarks, 2, "treatment", "responding")
return $a/name
```

This query returns all the names of all patients for which `tf:containsNearText` returns `true`. This is the case for all `remarks` nodes in which the words "treatment" and "responding" are within a distance of two word, i.e., at most one word token may be in-between. Given the sample data, Bloggs is the only patient.

## tf:containsText

Search for word tokens in a search string.

### Syntax

---

```
tf:containsText(node $input, string $searchString) => boolean
```

### Description

---

This text retrieval function searches a node for a string, consisting of one or more words and returns `true` if the node contains the word tokens in the specified order.

With `tf:containsText` you can perform search operations including the use of a wildcard character. The section *Pattern Matching* in the *XQuery 4 User Guide* explains this in detail.

### Arguments

---

#### **\$input**

node in which the search takes place

#### **\$searchString**

string containing a sequence of one or more words to be searched

### Examples

---

- Retrieve all patients who are responding to current treatment:

```
for $a in input()/patient
where tf:containsText($a/remarks, "responding to treatment")
return $a/name
```

This query returns all the names of all patients for which `tf:containsText` returns `true`. This is the case for all `remarks` nodes in which the word tokens "responding", "to" and "treatment" follow after one. Given the sample data, Bloggs is the only patient.

It is equivalent to the following query formulated using `tf:containsAdjacentText`, which also respects the token order and searches for the three tokens following directly one after another, using the distance value "1":

```
for $a in input()/patient
where tf:containsAdjacentText($a/remarks, 1, "responding", "to", "treatment")
return $a/name
```

- Retrieve all books about TCP/IP:

```
for $a in input()/bib/book
where tf:containsText($a/title, "TCP*IP")
return $a/title
```

## **tf:content-type**

Get the content type of the specified document.

### **Syntax**

---

```
tf:content-type(documentNode $doc) => string
```

### **Description**

---

This function returns the content type of the document that is identified by `$doc`.

### **Arguments**

---

**\$doc**  
document node



## tf:createAdjacentTextReference

Create reference descriptions for adjacent text locations.

### Syntax

---

```
tf:createAdjacentTextReference(node $input, integer $distance, token+ $word) => node*
```

### Description

---

The function `tf:createAdjacentTextReference` is specific to Tamino. It takes a node sequence, an integer value indicating a distance, and one or more word tokens as arguments. Similar to the function `tf:containsAdjacentText` it searches all word tokens of the node that are within the specified distance respecting the token order. However, it does not return a Boolean value, but a sequence of reference descriptions for all nodes found.

You can use reference descriptions for subsequent highlighting of these nodes, see [tf:highlight](#).

With `tf:createAdjacentTextReference` you can create reference descriptions based on a search using wildcard characters. The section *Pattern Matching* in the *XQuery 4 User Guide* explains this in detail.

### Arguments

---

**\$input**

node

**\$distance**

integer value denoting proximity distance

**\$word**

a word token

### Example

---

- Create reference descriptions in `remarks` nodes stating that patients are responding to current treatment:

```
for $a in input()/patient
return tf:createAdjacentTextReference($a/remarks, 3, "patient", "to", "treatment")
```

This query returns reference descriptions to the locations of all word tokens found:

```
<xq:result xmlns:xq="http://namespaces.softwareag.com/tamino/XQuery/result">
  <ino:object ino:collection="Hospital" ino:docid="2" ino:doctype="patient" ino:doctypeid="1" ino:end="7"
    ino:nodeid="87" ino:start="0" xmlns:ino="http://namespaces.softwareag.com/tamino/response2" />
  <ino:object ino:collection="Hospital" ino:docid="2" ino:doctype="patient" ino:doctypeid="1" ino:end="24"
    ino:nodeid="87" ino:start="22" xmlns:ino="http://namespaces.softwareag.com/tamino/response2" />
  <ino:object ino:collection="Hospital" ino:docid="2" ino:doctype="patient" ino:doctypeid="1" ino:end="35"
    ino:nodeid="87" ino:start="25" xmlns:ino="http://namespaces.softwareag.com/tamino/response2" />
</xq:result>
```

## tf:createNearTextReference

Create reference descriptions for nearby text locations.

### Syntax

---

```
tf:createNearTextReference(node $input, integer $distance, token+ $word) => node*
```

### Description

---

The function `tf:createNearTextReference` is specific to Tamino. It takes a node sequence, an integer value indicating a distance, and one or more word tokens as arguments. Similar to the function `tf:containsNearText` it searches all word tokens of the node that are within the specified distance regardless of the token order. However, it does not return a Boolean value, but a sequence of reference descriptions for all nodes found.

You can use reference descriptions for subsequent highlighting of these nodes, see [tf:highlight](#).

With `tf:createNearTextReference` you can create reference descriptions based on a search using wildcard characters. The section *Pattern Matching* in the *XQuery 4 User Guide* explains this in detail.

### Arguments

---

**\$input**

node

**\$distance**

integer value denoting proximity distance

**\$word**

a word token

### Example

---

- Create reference descriptions in `remarks` nodes stating that patients respond to treatment:

```
for $a in input()/patient
return tf:createNearTextReference($a/remarks, 2, "treatment", "responding")
```

This query returns reference descriptions to the locations of all word tokens found.

```
<xq:result xmlns:xq="http://namespaces.softwareag.com/tamino/XQuery/result">
  <ino:object ino:collection="Hospital" ino:docid="2" ino:doctype="patient" ino:doctypeid="1" ino:end="21"
    ino:nodeid="87" ino:start="11" xmlns:ino="http://namespaces.softwareag.com/tamino/response2" />
  <ino:object ino:collection="Hospital" ino:docid="2" ino:doctype="patient" ino:doctypeid="1" ino:end="35"
    ino:nodeid="87" ino:start="25" xmlns:ino="http://namespaces.softwareag.com/tamino/response2" />
</xq:result>
```

## tf:createNodeReference

Create reference descriptions for nodes.

### Syntax

```
tf:createNodeReference(node* $input) => node*
```

### Description

The function `tf:createNodeReference` is specific to Tamino. It takes a node sequence as argument and returns a sequence of reference descriptions for all nodes.

You can use reference descriptions for subsequent highlighting of these nodes, see [tf:highlight](#).

### Argument

#### `$input`

a sequence of nodes

### Example

- Create reference descriptions to all review nodes:

```
for $a in input()/reviews
return tf:createNodeReference($a/entry/review)
```

This query returns reference descriptions to all nodes found:

```
<xq:result xmlns:xq="http://namespaces.softwareag.com/tamino/XQuery/result">
  <ino:object ino:collection="XMP" ino:docid="1" ino:doctype="reviews" ino:doctypeid="1" ino:nodeid="12"
    xmlns:ino="http://namespaces.softwareag.com/tamino/response2" />
  <ino:object ino:collection="XMP" ino:docid="1" ino:doctype="reviews" ino:doctypeid="1" ino:nodeid="24"
    xmlns:ino="http://namespaces.softwareag.com/tamino/response2" />
  <ino:object ino:collection="XMP" ino:docid="1" ino:doctype="reviews" ino:doctypeid="1" ino:nodeid="36"
    xmlns:ino="http://namespaces.softwareag.com/tamino/response2" />
</xq:result>
```

## tf:createTextNode

Create a text node from a string.

### Syntax

---

```
tf:createTextNode(string $string) => textNode
```

### Description

---

The function `tf:createTextNode` is specific to Tamino. It takes a string as argument and creates a text node containing the value of the string.



**Note:** This function is deprecated and will be removed in future versions of Tamino. You should use [CompTextConstructor](#) instead. See the examples for details.

### Argument

---

**\$string**

string value

### Example

---

- Replace a book title directly on the text node:

```
update replace input()/bib/book/title[. = "TCP/IP Illustrated" ]/text()  
  with tf:createTextNode("TCP/IP Illustrated I. The Protocols")
```

The text node of the `title` element with the content "TCP/IP Illustrated" is replaced with a text node created by `tf:createTextNode` that has the contents "TCP/IP Illustrated I. The Protocols". Instead of this deprecated function you could use a computed text node constructor:

```
update replace input()/bib/book/title[. = "TCP/IP Illustrated" ]/text()  
  with text { "TCP/IP Illustrated I. The Protocols" }
```

Alternatively you could also write this query as follows:

```
update replace input()/bib/book/title[. = "TCP/IP Illustrated" ]/text()  
    with <title>TCP/IP Illustrated I. The Protocols</title>/text()
```

## tf:createTextReference

Create reference descriptions for text locations.

### Syntax

---

```
tf:createTextReference(node* $input, string $searchString) => node*
```

### Description

---

The function `tf:createTextReference` is specific to Tamino. It takes a node sequence and a search string as arguments. Similar to the function `tf:containsText` it searches a string, consisting of one or more words, in a node. However, it does not return a Boolean value, but a sequence of reference descriptions for each node found.

You can use reference descriptions for subsequent highlighting of these nodes, see [tf:highlight](#).

With `tf:createTextReference` you can create reference descriptions based on a search using wildcard characters. The section *Pattern Matching* in the *XQuery 4 User Guide* explains this in detail.

### Arguments

---

**\$input**

node

**\$searchString**

string value

### Examples

---

- Create reference descriptions to remarks for those patients who respond to current treatment:

```
for $a in input()/patient
return tf:createTextReference($a/remarks, "responding to treatment")
```

This query returns reference descriptions to the locations of all word tokens found:



```
<xq:result xmlns:xq="http://namespaces.softwareag.com/tamino/XQuery/result">
  <ino:object ino:collection="Hospital" ino:docid="2" ino:doctype="patient" ino:doctypeid="1" ino:end="21"
    ino:nodeid="87" ino:start="11" xmlns:ino="http://namespaces.softwareag.com/tamino/response2" />
  <ino:object ino:collection="Hospital" ino:docid="2" ino:doctype="patient" ino:doctypeid="1" ino:end="24"
    ino:nodeid="87" ino:start="22" xmlns:ino="http://namespaces.softwareag.com/tamino/response2" />
  <ino:object ino:collection="Hospital" ino:docid="2" ino:doctype="patient" ino:doctypeid="1" ino:end="35"
    ino:nodeid="87" ino:start="25" xmlns:ino="http://namespaces.softwareag.com/tamino/response2" />
</xq:result>
```

- Create reference descriptions to books for which the review text uses the word “discussion”:

```
for $a in input()/reviews
let $b := $a/entry/review
return tf:createTextReference($b, "discussion")
```

This query returns reference descriptions to the locations of all word tokens found; here in two different nodes:

```
<xq:result xmlns:xq="http://namespaces.softwareag.com/tamino/XQuery/result">
  <ino:object ino:collection="XMP" ino:docid="1" ino:doctype="reviews" ino:doctypeid="1" ino:end="22" ino:nodeid="12"
    ino:start="12" xmlns:ino="http://namespaces.softwareag.com/tamino/response2" />
  <ino:object ino:collection="XMP" ino:docid="1" ino:doctype="reviews" ino:doctypeid="1" ino:end="31" ino:nodeid="24"
    ino:start="21" xmlns:ino="http://namespaces.softwareag.com/tamino/response2" />
</xq:result>
```

- Create reference descriptions for the occurrences of "Kansas City" and "New York City":

```
let $x := <p>This is Kansas City and this is New York City</p>
for $d in $x//text()
let $a := (let $ref1 := tf:createTextReference($d, "Kansas City")
  for $refeach in $ref1
  return $refeach)
let $b := (let $ref2 := tf:createTextReference($d, "New York City")
  for $refeach in $ref2
  return $refeach)
return <output>{$a}{$b}</output>
```

Note that reference descriptions are generated for every token that the search term consists of. This is why there are five references in the output and not just two:

```
<xq:result xmlns:xq="http://namespaces.softwareag.com/tamino/XQuery/result">
  <output>
    <ino:object ino:docid="1" ino:doctypeid="0" ino:end="14" ino:nodeid="2" ino:start="8"
      xmlns:ino="http://namespaces.softwareag.com/tamino/response2" />
    <ino:object ino:docid="1" ino:doctypeid="0" ino:end="19" ino:nodeid="2" ino:start="15"
      xmlns:ino="http://namespaces.softwareag.com/tamino/response2" />
    <ino:object ino:docid="1" ino:doctypeid="0" ino:end="35" ino:nodeid="2" ino:start="32"
      xmlns:ino="http://namespaces.softwareag.com/tamino/response2" />
    <ino:object ino:docid="1" ino:doctypeid="0" ino:end="40" ino:nodeid="2" ino:start="36"
      xmlns:ino="http://namespaces.softwareag.com/tamino/response2" />
    <ino:object ino:docid="1" ino:doctypeid="0" ino:end="45" ino:nodeid="2" ino:start="41"
      xmlns:ino="http://namespaces.softwareag.com/tamino/response2" />
  </output>
</xq:result>
```

## tf:document

Document constructor.

### Syntax

---

```
tf:document(item* $content, string? $content-type) => documentNode
```

```
tf:document(item* $content, string? $content-type, element? $acl) => documentNode
```

### Description

---

It is possible to construct a non-XML document node using the document constructor: `document {$doc}`; in this case, the document type is assigned automatically. If you need to specify the document type and/or an ACL for the document, you should use this function.

### Arguments

---

#### **\$content**

The content of the document.

#### **\$content-type**

If non-empty, this argument specifies the content type of the newly-created document.

#### **\$acl**

If non-empty, this argument specifies the access control list for the document.

## tf:getCollation

Get collation information for a single node.

### Syntax

---

```
tf:getCollation(node $node) => string
```

### Description

---

The function `tf:getCollation` is specific to Tamino. It takes a node as argument and returns the collation specification for that node as a string.

### Argument

---

#### **\$node**

either element or attribute node

### Example

---

- Get collation information for the `surname` element:

```
let $a := input()/patient/name/surname
return tf:getCollation($a)
```

The returned string reflects the default settings when no collation had been defined in the schema:

```
http://www.softwareag.com/tamino/collation?language=en;strength=;caseFirst=;alternate=;caseLevel=;french=;normalization=
```

The key-value pairs after the question mark correspond to what you can define in a `tsd:collation` schema element.

- Get all patient names that are compared as equal to "Müller":

```
for $i in input()/patient/name
where compare($i/surname, 'Müller', tf:getCollation($i/surname)) = 0
return $i
```

This query returns those `name` elements whose `surname` child element has a contents that is compared equal to the string "Müller" using the collation information for `surname` elements.

## tf:getCollection

Get collection name for some node within.

### Syntax

---

```
tf:getCollection(documentNode $node) => NMTOKEN
```

### Description

---

The function `tf:getCollection` is specific to Tamino. It takes a node as argument and returns the name of the collection that the node is stored in.

### Argument

---

**\$node**

a single node of any kind

### Examples

---

- Get the name of the collection that contains at least one `book` element:

```
tf:getCollection(input()//book[1])
```

It would be a type exception at runtime, if you just try `tf:getCollection(input()//book)`, since the function expects a single node and not a node sequence.

- For all `title` elements get the collection name:

```
for $a in input()//title  
return tf:getCollection($a)
```

## **tf:get-current-user**

Get name of login user, if applicable.

### **Syntax**

---

```
tf:get-current-user() => string?
```

### **Description**

---

The function `tf:get-current-user` is specific to Tamino. It returns the name of the login user if present; otherwise it returns the empty sequence. If the database runs without security there is no login user.

## tf:getDocname

Get document name for document element.

### Syntax

---

```
tf:getDocname(documentNode $document) => string
```

### Description

---

The function `tf:getDocname` is specific to Tamino. It takes a document node as argument. It returns the document name of the node if present; otherwise it returns the empty sequence. This is a unique identifier since document names are unique in Tamino.

In general, you should apply the function `fn:root` to a node before passing it the `tf:getDocname`.

### Argument

---

**\$document**  
document node

### Examples

---

```
for $i in collection("ino:etc")/*  
return tf:getDocname(fn:root($i))
```



## tf:getInoId

Get ino:id for document element.

### Syntax

```
tf:getInoId(node $document) => unsignedInt
```

### Description

The function `tf:getInoId` is specific to Tamino. It takes a node as argument and returns the value of the public `ino:id` of the document that the node is part of. It returns 0 if the node is not stored in Tamino.

### Argument

**\$document**  
node

### Example

- Return the inoIds of all non-XML documents:

```
declare namespace ino="http://namespaces.softwareag.com/tamino/response2"  
for $a in collection("ino:etc")/ino:nonXML  
return tf:getInoId($a)
```

## tf:getLastModified

Get time stamp of last modification.

### Syntax

---

```
tf:getLastModified(documentNode $document) => dateTime
```

### Description

---

The function `tf:getLastModified` is specific to Tamino. It takes a document node as argument and returns the time stamp of its last modification. The time zone used is the Coordinated Universal Time (UTC), which is indicated by the letter "Z" at the end.

### Argument

---

**\$document**  
document node

### Example

---

- For all patient records return patient names and the time stamps of their last modification inside a new `patients` element:

```
for $a in input()  
return  
  <patients>  
    { $a/patient/name/* }  
    <lastChange>{ tf:getLastModified($a) }</lastChange>  
  </patients>
```

A typical response looks like this:

```
<xq:result xmlns:xq="http://namespaces.softwareag.com/tamino/XQuery/result">
  <patients>
    <surname>Atkins</surname>
    <firstname>Paul</firstname>
    <lastChange>2003-04-09T09:07:31Z</lastChange>
  </patients>
  <patients>
    <surname>Bloggs</surname>
    <firstname>Fred</firstname>
    <lastChange>2003-04-01T06:48:54Z</lastChange>
  </patients>
</xq:result>
```

## tf:highlight

Highlight text based on reference descriptions.

### Syntax

---

```
tf:highlight(node $node, node $description, string $markerString) => node*
```

### Description

---

The function `tf:highlight` is specific to Tamino. It takes a sequence of nodes, a sequence of reference descriptions and a marker string as arguments and returns `$node` along with any text or nodes, for which a reference description has been created, as enclosed between processing instructions (PIs).

The PIs are used to mark the highlighted text. They have the following format: The PI target is always `$markerString`. It is followed by either "+" or "-", indicating start and end of the highlighted token. The subsequent number is the current number of highlights in the result document.

If `$markerString` is empty, the string "Tamino-Highlighting" is used.

### Arguments

---

**`$node`**

node in which the search takes place

**`$description`**

a reference description

**`$markerString`**

a string value

### Examples

---

In the patient sample data, there is a `remarks` element for the patient Bloggs that reads: "Patient is responding to treatment. Dr. Shamir.". This remark should serve as example text here.

- Highlight book reviews using the word "discussion":

```
for $a in input()/reviews/entry
let $ref := tf:createTextReference($a/review, "discussion")
return tf:highlight($a, $ref, "REV_DISC")
```

This is the resulting document:

```
<entry>
  <title>Data on the Web</title>
  <price>34.95</price>
  <review>A very good <?REV_DISC + 1 ?>discussion<?REV_DISC - 1 ?> of ↵
semi-structured database systems and XML.</review>
</entry>
<entry>
  <title>Advanced Programming in the Unix environment</title>
  <price>65.95</price>
  <review>A clear and detailed <?REV_DISC + 2 ?>discussion<?REV_DISC - 2 ?> of ↵
UNIX programming.</review>
</entry>
<entry>
  <title>TCP/IP Illustrated</title>
  <price>65.95</price>
  <review>One of the best books on TCP/IP.</review>
</entry>
```

Note that all entry nodes are returned, although only for the first two carry highlighted text. If only nodes with highlighted text should be returned, simply add a `where` clause checking the Boolean return value of the reference creator function:

```
for $a in input()/reviews/entry
let $ref := tf:createTextReference($a/review, "discussion")
where $ref
return tf:highlight($a, $ref, "REV_DISC")
```

## tf:narrowerTerm

Search for immediate subordinate term.

## Syntax

---

```
tf:narrowerTerm(string $termName, string? $thesaurus) => unspecified
```

## Description

---

The function `tf:narrowerTerm` is specific to Tamino. It takes a string as argument which is the name of a thesaurus entry. You can supply a further optional string value that specifies the name of a thesaurus. In the scope of text retrieval functions this function returns the immediate subordinate term of `$termName`. If there is no subordinate term defined, only `$termName` is returned.

This function can only be used within the scope of the following functions:

```
tf:containsAdjacentText  
tf:containsNearText  
tf:containsText  
tf:createAdjacentTextReference  
tf:createNearTextReference  
tf:createTextReference
```

## Argument

---

### **\$termName**

name of a thesaurus entry that is defined in the `termName` element

### **\$thesaurus**

name of a thesaurus

## Examples

---

- Return subordinate term of "carnivore":

```
declare namespace ino="http://namespaces.softwareag.com/tamino/response2"
for   $p in input()/ino:term
where tf:containsText($p/ino:termName, tf:narrowerTerm("carnivore"))
return $p/ino:termName
```

## tf:narrowerTerms

Search for all subordinate terms.

### Syntax

---

```
tf:narrowerTerms(string $termName, string? $thesaurus) => unspecified
```

### Description

---

The function `tf:narrowerTerms` is specific to Tamino. It takes a string as argument which is the name of a thesaurus entry. You can supply a further optional string value that specifies the name of a thesaurus. In the scope of text retrieval functions this function returns all subordinate terms of `$termName`. If there is no subordinate term defined, only `$termName` is returned.

This function can only be used within the scope of the following functions:

```
tf:containsAdjacentText  
tf:containsNearText  
tf:containsText  
tf:createAdjacentTextReference  
tf:createNearTextReference  
tf:createTextReference
```

### Argument

---

**\$termName**

name of a thesaurus entry that is defined in the `termName` element

**\$thesaurus**

name of a thesaurus

### Examples

---

- Return all subordinate terms of "carnivore":



```
declare namespace ino="http://namespaces.softwareag.com/tamino/response2"
for   $p in input()/ino:term
where tf:containsText($p/ino:termName, tf:narrowerTerms("carnivore"))
return $p/ino:termName
```

## tf:nonXML-kind

Get the kind of a non-XML document.

### Syntax

---

```
tf:nonXML-kind(document-node $doc) => string?
```

### Description

---

The function `tf:nonXML-kind` returns a string that describes the kind of the given document. The returned value is one of the following:

**“text”**

if `$doc` is a text non-XML document;

**“binary”**

if `$doc` is a binary non-XML document;

**() (the empty sequence)**

if neither of the above applies. This is the case if `$doc` is a pure XML document.

### Argument

---

**\$doc**

The document that is to be checked.

## tf:parse

Convert non-XML content to XML.

### Syntax

---

```
tf:parse(string $input) => document-node
```

### Description

---

This function returns a document node whose content is the result of applying an XML parser to `$input`.

### Argument

---

#### **\$input**

The non-XML content to be converted.

## tf:phonetic

Search text based on phonetic similarities.

## Syntax

```
tf:phonetic(string $searchString) => unspecified
```

## Description

The function `tf:phonetic` is specific to Tamino. It takes a search string as argument and returns all strings that are “phonetically equivalent”. It can only be used within the scope of the following functions:

```
tf:containsAdjacentText
tf:containsNearText
tf:containsText
tf:createAdjacentTextReference
tf:createNearTextReference
tf:createTextReference
```

Tamino performs this search according to a set of rules that is modeled after the widely known **Soundex algorithm**. It is based on the pronunciation of the English language, but includes also checks for character combinations that occur in German. This means that the accuracy of the algorithm is highest for English and German, but it can also be used for other languages. However, it is not exact: Sometimes it will fail to identify words that are homophones, and sometimes the algorithm will incorrectly detect a match when in fact the pronunciation of the word is quite distinct. The algorithm works by reducing letters or combinations of letters to their phonetic equivalents according to the following rules:

Letters	Phonetic Equivalent
A, E, I, O, U, Y (initial position)	A
P, B	B
F, V, W, (P + H)	F
G, K, Q, (C + [A, E, H, I, J, K, L, O, Q, R, U, X, Y])	G
L	L
M	M
N	N
R	R
C, S, Z, (D + [C, S, Z]), (X + [C, K, Q]), (T + [C, S, Z]), (S+C), (Z+C)	S

Letters	Phonetic Equivalent
D, T	D
(G + G + S)	GS
(X - [C, K, Q], *)	GS
H	(ignored)

Here, "+" denotes two letters appearing in the order shown. "[...]" denotes alternative letters, "-" denotes exclusion, i.e., two letters appearing together of which the second letter is not one of the letters listed. Finally, "\*" denotes any letter.

So "(X + [C, K, Q])" means a sequence of letters consisting of "X" followed by one of the letters "C", "K" or "Q", whereas "(X-[C, K, Q], \*)" means a sequence of three letters consisting of "X" followed by any letter other than "C" or "K" or "Q", followed by any letter.

More elaborated rules take precedence over simple rules: For example, if a word contains the adjacent letters "P" and "H", the rule reducing the combination of "(P + H)" to "F" has precedence over the two simple rules that reduce "P" to "B" and ignore "H".

Example: "PHONETIC" is interpreted as "(P + H), (O), (N), (E), (T), (I), (C)" and reduced to "FNDS".



**Note:** The value of the server parameter "markup as delimiter" is respected when determining the word tokens. See the documentation of the Tamino Manager for details.

## Arguments

**\$searchString**  
string value

## Example

- Retrieve the names of all patients whose surname sound like "Meier".

```
for $a in input()/patient
where tf:containsText($a/name/surname, tf:phonetic("Meier"))
return $a/name
```

This query effectively retrieves all patient names that are written as "Meier", "Maier", "Mayer", or "Meyer" as they all sound alike.

## tf:query

Execute an XQuery

### Syntax

---

```
tfquery(string $query) => item*
```

### Description

---

Function `tf:query` compiles and executes a nested XQuery, that is passed as a string. It returns the result of the query as a sequence of items. Any errors that occur during compilation or execution of the nested query will be reported as execution errors of the calling query.

### Parameters

---

#### **\$query**

A string that contains an executable query.

### Examples

---

- The example uses a static query which might just as well been executed directly, but the power of this function is in executing queries that are created during run-time, for example by reading it from somewhere or assembling it from pieces.

```
let $query := "for $book in collection('ino:etc')//book
where $book//last='Buneman'
return $book"
return tf:query($query)
```

## **tf:serialize**

Convert XML content to non-XML.

### **Syntax**

---

```
tf:serialize(item* $input) => string
```

### **Description**

---

This function returns a string that contains the serialization of the input.

### **Argument**

---

#### **\$input**

The XML content that is to be converted to non-XML.

## **tf:setDocname**

Modify name of a document.

### **Syntax**

---

```
tf:setDocname(node $node, string? $name)
```

### **Description**

---

Modifies the name of an existing document.

### **Arguments**

---

#### **\$node**

The document node.

#### **\$name**

The new name to be assigned to the document. If this argument is the empty sequence, the docname is removed.



## tf:stem

Search text based on word stems.

## Syntax

```
tf:stem(string $searchString) => unspecified
```

## Description

The function `tf:stem` is specific to Tamino. It takes a search string as argument and returns all strings that share the same stem as the search string. It can only be used within the scope of the following functions:

```
tf:containsAdjacentText  
tf:containsNearText  
tf:containsText  
tf:createAdjacentTextReference  
tf:createNearTextReference  
tf:createTextReference
```

Determining the word tokens that have the same word stem as the search string requires language-specific information. Currently, the pre-defined stemming information is only suitable for German.



### Notes:

1. For better performance, Tamino uses a special stemming index that must be activated for the current database. Therefore, you must set the database server parameter option "stemming index" to "yes". See the documentation of the Tamino Manager for details.
2. The value of the server parameter "markup as delimiter" is respected when determining the word tokens. See the documentation of the Tamino Manager for details.

## Argument

### **\$searchString**

a string value

## Example

---

- In the paragraphs of some chapter, retrieve all occurrences of the German word "Bank" in the sense of a bank dealing with money:

```
let $text :=
  <chapter>
    <para>Die Bank eröffnete drei neue Filialen im Verlauf der letzten fünf ↵
Jahre.</para>
    <para>Ermüdet von dem Spaziergang setzte sich die alte Dame erleichtert auf die ↵
gepflegt
    wirkende Bank mitten im Stadtpark.</para>
    <para>Die aktuelle Bilanz der Bank zeigt einen Anstieg der liquiden Mittel im ↵
Vergleich
    zum Vorjahresquartal.</para>
  </chapter>
for $a in $text//para
let $check :=
  for $value in ("Geld", "Bilanz", "Filiale", "monetär", "Aktie")
  return tf:containsNearText($a, 10, tf:stem($value), tf:stem("Bank"))
where count($check[. eq true()]) > 0
return $a
```

A sequence creates a word family that is valid for one of two readings of the German word "bank". For each of these related words it is checked whether the current paragraph contains an inflected form that is no longer than ten unmatched word tokens apart from an inflected form of the word "bank". The second `let` clause returns a sequence of five Boolean values. If at least one of them is true—expressed by the `where` clause—the corresponding `para` element is returned as part of the result.

## tf:synonym

Search for synonym terms.

### Syntax

---

```
tf:synonym(string $termName, string? $thesaurus) => unspecified
```

### Description

---

The function `tf:synonym` is specific to Tamino. It takes a string as argument which is the name of a thesaurus entry. The function returns all synonymous terms of `$termName`. If there are no synonyms defined, only `$termName` is returned.

This function can only be used within the scope of the following functions:

```
tf:containsAdjacentText  
tf:containsNearText  
tf:containsText  
tf:createAdjacentTextReference  
tf:createNearTextReference  
tf:createTextReference
```

### Arguments

---

**`$termName`**

name of a thesaurus entry that is defined in the `termName` element

**`$thesaurus`**

name of a thesaurus

### Examples

---

- Return synonymous terms of "dog" in the specified document:

```
let $doc := <doc>
  <p>Have you seen the large dog around the corner?</p>
  <p>On the farm nearby, a checkered whelp was playing on the ground with some ↵
cats.</p>
  <p>Also, some horses could be seen in the stable.</p>
</doc>
for   $p in $doc/p
where tf:containsText($p, tf:synonym("dog"))
return $p
```

## tf:text-content

Retrieve non-XML text content from a non-XML document.

### Syntax

```
tf:text-content(documentNode $doc) => string?
```

### Description

The content of a persistent pure non-XML text document can be accessed via its XML view; however, this does not work for a non-XML text document that has a shadow XML document, because the shadow XML document obstructs the view of the text content. Also any text content of a transient non-XML text document is not accessible this way, because the root element is not present for a transient non-XML document. Use the function `tf:text-content` to retrieve non-XML text content.

The function returns:

- the non-XML text content; if `$doc` is a text non-XML document;
- the empty sequence, if `$doc` is a binary non-XML document;.
- otherwise, a type error.

### Argument

**\$doc**

The document

### Example

```
tf:text-content(document{"ABC"})
```

Returns the string "ABC" that was previously constructed as a non-XML text content.

## xs:anyURI

Construct an `anyURI` value from an item value.

### Syntax

---

```
xs:anyURI(item $value) => anyURI
```

### Description

---

This is a constructor function that takes an item value as argument and returns a value of type `xs:anyURI`. If the argument is a literal, then the literal must be a valid lexical form of its type.

As specified in the respective section of the W3C recommendation [XML Schema Part 2: Datatypes](#), a value of type `xs:anyURI` represents a unified resource identifier. Although possible it is not recommended to use space characters in `anyURI` values.

### Argument

---

**\$value**  
string value

### Example

---

- Construct a valid values of type `xs:anyURI`:

```
xs:anyURI("http://www.w3.org/2001/XMLSchema")
```

## **xs:base64Binary**

Construct a `base64Binary` value from an item value.

### **Syntax**

---

```
xs:base64Binary(item $value) => base64Binary
```

### **Description**

---

This is a constructor function that takes an item value as argument and returns a value of type `xs:base64Binary`. If the argument is a literal, then the literal must be a valid lexical form of its type.

As specified in the respective section of the W3C recommendation [XML Schema Part 2: Datatypes](#), a value of type `xs:base64Binary` represents Base64-encoded binary data. If the string argument contains characters outside the Base 64 alphabet, a type exception is raised. See section 6.8 of [RFC 2045](#) for information about the allowed characters.

### **Argument**

---

**\$value**  
string value

## xs:boolean

Construct a `boolean` value from an item value.

### Syntax

---

```
xs:boolean(item $value) => boolean
```

### Description

---

This is a constructor function that takes an item value as argument and returns a value of type `xs:boolean`. If the argument is a literal, then the literal must be a valid lexical form of its type.

As specified in the respective section of the W3C recommendation [XML Schema Part 2: Datatypes](#), a value of type `xs:boolean` represents a value to denote binary-valued logic. The lexical representation for "true" is the set of values {"true", 1}; for false it is the set of values {"false", 0}.

### Argument

---

**\$value**

item value

### Examples

---

- Generate the boolean value "true":

```
xs:boolean("1")
```

```
xs:boolean("true")
```

- This query generates an error since the argument is interpreted as a path expression:



```
xs:boolean(true)
```

## xs:byte

Construct a byte value from an item value.

### Syntax

---

```
xs:byte(item $value) => byte
```

### Description

---

This is a constructor function that takes an item value as argument and returns a value of type `xs:byte`. If the argument is a literal, then the literal must be a valid lexical form of its type.

As specified in the respective section of the W3C recommendation [XML Schema Part 2: Datatypes](#), `xs:byte` is a subtype derived from `xs:short`. The lexical representation is the set of numerical values in the range from -128 to 127. If no sign is used, "+" is assumed.

### Argument

---

**\$value**

item value

### Examples

---

- The following queries construct valid values of type `xs:byte`:

```
xs:byte(44)
```

```
xs:byte(-23)
```

```
xs:byte(-0)
```

- The following queries do not construct valid values of type `xs:byte`:

```
xs:byte(4400)
```

```
xs:byte(-223)
```

```
xs:byte(-+0)
```

## xs:date

Construct a `date` value from an item value.

### Syntax

---

```
xs:date(item $value) => date
```

### Description

---

This is a constructor function that takes an item value as argument and returns a value of type `xs:date`. If the argument is a literal, then the literal must be a valid lexical form of its type.

As specified in the respective section of the W3C recommendation [XML Schema Part 2: Datatypes](#), a value of type `xs:date` represents a Gregorian calendar date as defined in ISO 8601. The lexical form is CCYY-MM-DD where "CC" represents the century, "YY" the year, "MM" the month and "DD" the day. You may not truncate digits on the left side. To represent a date before Christ, you must prepend a minus sign.

### Argument

---

**\$value**

item value

### Examples

---

- Generate a valid value of type `xs:date`:

```
xs:date("2002-12-07")
```

- Add an entry for the first Roman emperor:

```
<emperor>
  <name>Augustus</name>
  <birth>
    <date>{ xs:date("-0063-09-23") }</date>
  </birth>
  <death>
    <date>{ xs:date("0014-08-14") }</date>
    <place>Nola</place>
  </death>
</emperor>
```

## xs:dateTime

Construct a `dateTime` value from an item value.

### Syntax

---

```
xs:dateTime(item $value) => dateTime
```

### Description

---

This is a constructor function that takes an item value as argument and returns a value of type `xs:dateTime`. If the argument is a literal, then the literal must be a valid lexical form of its type.

As specified in the respective section of the W3C recommendation [XML Schema Part 2: Datatypes](#), a value of type `xs:dateTime` represents a specific instant of time by a combination of date and time of day values. The lexical form is CCYY-MM-DDThh:mm:ss where "CC" represents the century, "YY" the year, "MM" the month and "DD" the day. It can be preceded by an optional leading minus sign to indicate a negative number. You may not truncate digits on the left side. The letter "T" separates date from time and "hh", "mm", "ss" represent hour, minute and second. Fractions of a second are appended with a leading ".".

A datetime value may include a time zone which is either a capital "Z" to denote Coordinated Universal Time (UTC), or it is a value whose lexical form is hh:mm, preceded by a minus or plus sign to indicate the difference from UTC.

### Argument

---

**\$value**  
item value

### Example

---

- Generate a valid value of type `xs:dateTime`:

```
xs:dateTime("2002-12-07T12:20:46.275+01:00")
```

This represents December 7, 2002 at 12:20:46 and 275/1000 seconds CET. The timezone CET is 1 hour after UTC.

## xs:decimal

Construct a `decimal` value from an item value.

### Syntax

---

```
xs:decimal(item $value) => decimal
```

### Description

---

This is a constructor function that takes an item value as argument and returns a value of type `xs:decimal`. If the argument is a literal, then the literal must be a valid lexical form of its type.

As specified in the respective section of the W3C recommendation [XML Schema Part 2: Datatypes](#), a value of type `xs:decimal` represents a decimal number. However, omitting a leading "0" in front of a fractional number is not allowed.

### Argument

---

**\$value**

item value

### Example

---

- Generate a valid value of type `xs:decimal`:

```
xs:decimal("002002.270")
```



## **xs:double**

Construct a `double` value from an item value.

### **Syntax**

```
xs:double(item $value) => double
```

### **Description**

This is a constructor function that takes an item value as argument and returns a value of type `xs:double`. If the argument is a literal, then the literal must be a valid lexical form of its type.

As specified in the respective section of the W3C recommendation [XML Schema Part 2: Datatypes](#), a value of type `xs:double` represents a double-precision floating point number according to IEEE 754-1985. The lexical form uses scientific notation, consisting of the mantissa which must be a decimal number, the letter "E" or "e", and an exponent which must be an integer value.

### **Argument**

**\$value**  
item value

### **Example**

- Generate a valid value of type `xs:double`:

```
xs:double("002002.270")
```

This number is represented as `2.00227E3`.

## xs:duration

Construct a `duration` value from an item value.

## Syntax

---

```
xs:duration(item $value) => duration
```

## Description

---

This is a constructor function that takes an item value as argument and returns a value of type `xs:duration`. If the argument is a literal, then the literal must be a valid lexical form of its type.

As specified in the respective section of the W3C recommendation [XML Schema Part 2: Datatypes](#), a value of type `xs:duration` represents a duration of time. Its lexical form is `PnYnMnDTnh nmns`, where *nY* is the number of years, *nM* is the number of months, *nD* is the number of days, *T* is the separator between date and time, *nH* is the number of hours, *nM* is the number of minutes and *nS* is the number of seconds. Fractions of a second are appended with a leading ".".

You can omit any of the numbers, if they equal zero. However, at least one number and its designator must be present. If you do not specify a time part, the separator character "T" must not be present. The leading "P" must always be present.

You can specify a negative duration by prepending a minus sign. If no sign is used, "+" is assumed.

## Argument

---

**\$value**

item value

## Examples

---

- Generate an entry for J.S. Bach's St. Matthew passion and its duration:

```
<opus number="BWV244">  
  <title>Matthäus-Passion</title>  
  <duration>xs:duration("PT170M52S")</duration>  
</opus>
```

- These queries create further valid duration values:

```
xs:duration("P30Y243D")
```

```
xs:duration("PT5H")
```

```
xs:duration("-P30M")
```

- These queries create invalid duration values:

```
xs:duration("P30Y243DT")
```

```
xs:duration("P5H")
```

```
xs:duration("P-30M")
```

## xs:ENTITY

Construct an ENTITY value from an item value.

### Syntax

---

```
xs:ENTITY(item $value) => ENTITY
```

### Description

---

This is a constructor function that takes an item value as argument and returns a value of type `xs:ENTITY`. If the argument is a literal, then the literal must be a valid lexical form of its type.

As specified in the respective section of the W3C recommendation [XML Schema Part 2: Datatypes](#), a value of type `xs:ENTITY` denotes an entity, as defined in the [XML 1.0](#) recommendation. It is derived from the type `xs:NCName` and the lexical form of `xs:ENTITY` is the set of all strings that match the production of [NCName](#).

### Argument

---

**\$value**  
item value

### Examples

---

- The following queries generate valid values of type `xs:ENTITY`

```
xs:ENTITY("_header")
```

```
xs:ENTITY("R2D2")
```

- The following queries generate invalid values of type `xs:ENTITY`

```
xs:ENTITY("-header")
```

```
xs:ENTITY("1A")
```

## xs:float

Construct a `float` value from an item value.

### Syntax

---

```
xs:float(item $value) => float
```

### Description

---

This is a constructor function that takes an item value as argument and returns a value of type `xs:float`. If the argument is a literal, then the literal must be a valid lexical form of its type.

As specified in the respective section of the W3C recommendation [XML Schema Part 2: Datatypes](#), a value of type `xs:float` represents a single-precision floating point number according to IEEE 754-1985. The lexical form uses scientific notation, consisting of the mantissa which must be a decimal number, the letter "E" or "e", and an exponent which must be an integer value.

### Argument

---

**\$value**  
item value

### Example

---

- Generate a valid value of type `xs:float`:

```
xs:float("002002.270")
```

This number is represented as 2.00227E3.

## xs:gDay

Construct a `gDay` value from an item value.

### Syntax

```
xs:gDay(item $value) => gDay
```

### Description

This is a constructor function that takes an item value as argument and returns a value of type `xs:gDay`. If the argument is a literal, then the literal must be a valid lexical form of its type.

As specified in the respective section of the W3C recommendation [XML Schema Part 2: Datatypes](#), a value of type `xs:gDay` represents a recurring Gregorian day as defined in ISO 8601. The lexical form is `---DD`, where "DD" denotes the day. You can append an optional time zone identifier. You may not omit any of the dashes in front.

### Argument

**\$value**

item value

### Example

- This query generates valid values of type `xs:gDay`:

```
xs:gDay("---27")
```

```
xs:gDay("---27Z")
```

```
xs:gDay("---27+01:00")
```

The last two queries use optional time zone identifiers: "Z" denoting UTC, and "+01:00" one hour ahead UTC.

## xs:gMonth

Construct a `gMonth` value from an item value.

### Syntax

---

```
xs:gMonth(item $value) => gMonth
```

### Description

---

This is a constructor function that takes an item value as argument and returns a value of type `xs:gMonth`. If the argument is a literal, then the literal must be a valid lexical form of its type.

As specified in the respective section of the W3C recommendation [XML Schema Part 2: Datatypes](#), a value of type `xs:gMonth` represents a recurring Gregorian month as defined in ISO 8601. The lexical form is `--MM--`, where "MM" denotes the month. You can append an optional time zone identifier. You may not omit any of the dashes.

### Argument

---

**\$value**  
item value

### Example

---

- This query generates valid values of type `xs:gMonth`:

```
xs:gMonth("--04--")
```

```
xs:gMonth("--04--Z")
```

```
xs:gMonth("--04---01:00")
```

The last two queries use optional time zone identifiers: "Z" denoting UTC, and "-01:00" one hour past UTC.



## xs:gMonthDay

Construct a `gMonthDay` value from an item value.

### Syntax

```
xs:gMonthDay(item $value) => gMonthDay
```

### Description

This is a constructor function that takes an item value as argument and returns a value of type `xs:gMonthDay`. If the argument is a literal, then the literal must be a valid lexical form of its type.

As specified in the respective section of the W3C recommendation [XML Schema Part 2: Datatypes](#), a value of type `xs:gMonthDay` represents a recurring Gregorian day of the month as defined in ISO 8601. The lexical form is `--MM-DD`, where "MM" denotes the month, and "DD" denotes the day. You can append an optional time zone identifier. You may not omit any of the dashes.

### Argument

**\$value**

item value

### Example

- This query generates valid values of type `xs:gMonthDay`:

```
xs:gMonthDay(" - -04-23")
```

```
xs:gMonthDay(" - -04-23Z")
```

```
xs:gMonthDay(" - -04-23-01:00")
```

The last two queries use optional time zone identifiers: "Z" denoting UTC, and "-01:00" one hour past UTC.

## xs:gYear

Construct a `gYear` value from an item value.

### Syntax

---

```
xs:gYear(item $value) => gYear
```

### Description

---

This is a constructor function that takes an item value as argument and returns a value of type `xs:gYear`. If the argument is a literal, then the literal must be a valid lexical form of its type.

As specified in the respective section of the W3C recommendation [XML Schema Part 2: Datatypes](#), a value of type `xs:gYear` represents a whole Gregorian year as defined in ISO 8601. The lexical form is CCYY, where "CC" represents the century and "YY" represents the year. You can use more digits in order to represent years outside the range 0000-9999, and you can also prepend a minus sign. You can also append an optional time zone identifier.

### Argument

---

**\$value**

item value

### Example

---

- This query generates valid values of type `xs:gYear`:

```
xs:gYear("1999")
```

```
xs:gYear("-0000")
```

```
xs:gYear("12403Z")
```

## xs:gYearMonth

Construct a `gYearMonth` value from an item value.

### Syntax

---

```
xs:gYearMonth(item $value) => gYearMonth
```

### Description

---

This is a constructor function that takes an item value as argument and returns a value of type `xs:gYearMonth`. If the argument is a literal, then the literal must be a valid lexical form of its type.

As specified in the respective section of the W3C recommendation [XML Schema Part 2: Datatypes](#), a value of type `xs:gYearMonth` represents a specific Gregorian month of the year as defined in ISO 8601. The lexical form is CCYY-MM, where "MM" denotes the century, "YY" denotes the year, and "MM" denotes the month. You can append an optional time zone identifier. You may not omit any of the dashes.

### Argument

---

**\$value**

item value

### Example

---

- This query generates valid values of type `xs:gYearMonth`:

```
xs:gYearMonth("1999-09")
```

```
xs:gYearMonth("-0000-09-09:00")
```

```
xs:gYearMonth("12403-03Z")
```

## xs:hexBinary

Construct a `hexBinary` value from an item value.

### Syntax

---

```
xs:hexBinary(item $value) => hexBinary
```

### Description

---

This is a constructor function that takes an item value as argument and returns a value of type `xs:hexBinary`. If the argument is a literal, then the literal must be a valid lexical form of its type.

As specified in the respective section of the W3C recommendation [XML Schema Part 2: Datatypes](#), a value of type `xs:hexBinary` represents binary data as hexadecimal digits.

### Argument

---

**\$value**  
item value

### Example

---

- Represent the header of files in JFIF used for JPEG graphics files:

```
<format>
  <name>JFIF</name>
  <header name="SOI Marker">{ xs:hexBinary("FFD8") }</header>
</format>
```

## xs:ID

Construct an ID value from an item value.

### Syntax

```
xs:ID(item $value) => ID
```

### Description

This is a constructor function that takes an item value as argument and returns a value of type `xs:ID`. If the argument is a literal, then the literal must be a valid lexical form of its type.

As specified in the respective section of the W3C recommendation [XML Schema Part 2: Datatypes](#), a value of type `xs:ID` denotes an ID attribute type, as defined in the [XML 1.0](#) recommendation. It is derived from the type `xs:NCName` and the lexical form of `xs:ID` is the set of all strings that match the production of [NCName](#).

For compatibility reasons values of this type should only be used as attribute values.

### Argument

**\$value**  
item value

### Example

- The following queries generate valid values of type `xs:ID`

```
xs:ID("_header")
```

```
xs:ID("R2D2")
```

- The following queries generate invalid values of type `xs:ID`

```
xs:ID("-header")
```

```
xs:ID("1A")
```



## xs:IDREF

Construct an IDREF value from an item value.

### Syntax

```
xs:IDREF(item $value) => IDREF
```

### Description

This is a constructor function that takes an item value as argument and returns a value of type `xs:IDREF`. If the argument is a literal, then the literal must be a valid lexical form of its type.

As specified in the respective section of the W3C recommendation [XML Schema Part 2: Datatypes](#), a value of type `xs:IDREF` denotes an ID attribute type, as defined in the [XML 1.0](#) recommendation. It is derived from the type `xs:NCName` and the lexical form of `xs:IDREF` is the set of all strings that match the production of [NCName](#).

For compatibility reasons values of this type should only be used as attribute values.

### Argument

**\$value**  
item value

### Examples

- The following queries generate valid values of type `xs:IDREF`

```
xs:IDREF("_header")
```

```
xs:IDREF("R2D2")
```

- The following queries generate invalid values of type `xs:IDREF`

```
xs:IDREF("-header")
```

```
xs:IDREF("1A")
```

## **xs:int**

Construct an `int` value from an item value.

### **Syntax**

```
xs:int(item $value) => int
```

### **Description**

This is a constructor function that takes an item value as argument and returns a value of type `xs:int`. If the argument is a literal, then the literal must be a valid lexical form of its type.

As specified in the respective section of the W3C recommendation [XML Schema Part 2: Datatypes](#), a value of type `xs:int` represents an integer value derived from `xs:long`. Please refer to the section *Derived built-in data types* in the *Tamino XML Schema User Guide* for information about the supported range of values.

### **Argument**

**\$value**  
item value

### **Example**

- Generate valid values of type `xs:int`:

```
xs:int("002002.270")
```

```
xs:int("002002")
```

The first invocation of `xs:int` raises a type exception, whereas the argument for the second invocation is valid. That number is represented as 2002.

## xs:integer

Construct an `integer` value from a value of any simple type.

### Syntax

---

```
xs:integer(anySimpleType $value) => integer
```

### Description

---

This is a constructor function that takes a value of any simple type as argument and returns a value of type `xs:integer`. If the argument is a literal, then the literal must be a valid lexical form of its type.

As specified in the respective section of the W3C recommendation [XML Schema Part 2: Datatypes](#), a value of type `xs:integer` represents an integer value derived from `xs:decimal`. Please refer to the section *Derived built-in data types* in the *Tamino XML Schema User Guide* for information about the supported range of values. Any decimal places after the point are truncated when casting.

Please note that a value of this type is not checked against the set of existing language identifiers.

### Argument

---

**\$value**

value of any simple type

### Example

---

- Generate valid values of type `xs:integer`:

```
xs:integer(2.00272E3)
```

```
xs:integer("002002")
```

The first invocation of `xs:integer` casts the floating-point number to an integer, while the second takes the string literal which is a valid lexical form of an integer value. In both cases the integer number is represented as 2002.

## xs:language

Construct a `language` value from an `item` value.

### Syntax

```
xs:language(item $value) => language
```

### Description

This is a constructor function that takes an `item` value as argument and returns a value of type `xs:language`. If the argument is a literal, then the literal must be a valid lexical form of its type.

As specified in the respective section of the W3C recommendation [XML Schema Part 2: Datatypes](#), a value of type `xs:language` represents a natural language identifier as defined by [RFC1766](#). The lexical representation is the set of all strings that are valid language identifiers as listed in the section *Language and Country Codes* in the *Tamino XML Schema User Guide*.

### Argument

**\$value**  
item value

### Example

- Create a citation and classify its text as British English:

```
let $lang := xs:language("en-GB")
return
<citation>
  <author><surname>Chomsky</surname><firstname>Noam</firstname></author>
  <text>{ attribute lang { $lang }
    }Colourless green ideas sleep furiously.</text>
</citation>
```

## xs:long

Construct a `long` value from an item value.

### Syntax

---

```
xs:long(item $value) => long
```

### Description

---

This is a constructor function that takes an item value as argument and returns a value of type `xs:long`. If the argument is a literal, then the literal must be a valid lexical form of its type.

As specified in the respective section of the W3C recommendation [XML Schema Part 2: Datatypes](#), `xs:byte` is a subtype derived from `xs:integer`. Please refer to the section *Derived built-in data types* in the *Tamino XML Schema User Guide* for information about the supported range of values.

### Argument

---

**\$value**

item value

### Example

---

- The following queries construct valid values of type `xs:long`:

```
xs:long(9223372036854775807)
```

```
xs:long(-23)
```

- The following query results in an error, since the specified numerical value is out of range:

```
xs:long(9223372036854775808)
```

## xs:Name

Construct a `Name` value from an item value.

## Syntax

---

```
xs:Name(item $value) => Name
```

## Description

---

This is a constructor function that takes an item value as argument and returns a value of type `xs:Name`. If the argument is a literal, then the literal must be a valid lexical form of its type.

As specified in the respective section of the W3C recommendation [XML Schema Part 2: Datatypes](#), a value of type `xs:Name` denotes an XML name, as defined in the [XML 1.0](#) recommendation. It is derived from the type `xs:token` and the lexical form of `xs:Name` is the set of all strings that match the production of `Name`.

## Argument

---

**\$value**  
item value

## Example

---

- The following queries generate valid values of type `xs:Name`:

```
xs:Name("_header")
```

```
xs:Name("R2D2:C3P0")
```

- The following queries generate invalid values:



```
xs:Name("-header")
```

```
xs:Name("1A")
```

## xs:NCName

Construct an `NCName` value from an item value.

### Syntax

---

```
xs:NCName(item $value) => NCName
```

### Description

---

This is a constructor function that takes an item value as argument and returns a value of type `xs:NCName`. If the argument is a string literal, then the literal must be a valid lexical form of its type.

As specified in the respective section of the W3C recommendation [XML Schema Part 2: Datatypes](#), a value of type `xs:NCName` denotes a “non-colonized” XML name, as defined in the [XML 1.0](#) recommendation. It is derived from the type `xs:Name` and the lexical form of `xs:Name` is the set of all strings that match the production of `Name`. Practically, a value of type `xs:NCName` is the same as of type `xs:Name` with the only exception that it may not contain a colon.

### Argument

---

**\$value**  
item value

### Examples

---

- The following queries generate valid values of type `xs:NCName`

```
xs:NCName( "_header" )
```

```
xs:NCName( "R2D2" )
```

- The following queries generate invalid values:

```
xs:NCName("-header")
```

```
xs:NCName("R2D2:C3PO")
```

## xs:NMTOKEN

Construct an NMTOKEN value from an item value.

### Syntax

---

```
xs:NMTOKEN(item $value) => NMTOKEN
```

### Description

---

This is a constructor function that takes an item value as argument and returns a value of type `xs:NMTOKEN`. If the argument is a literal, then the literal must be a valid lexical form of its type.

As specified in the respective section of the W3C recommendation [XML Schema Part 2: Datatypes](#), a value of type `xs:NCName` denotes an XML name token, as defined in the [XML 1.0](#) recommendation. It is derived from the type `xs:token` and the lexical form of `xs:NMTOKEN` is the set of all strings that match the production of `Nmtoken`. Practically, a value of type `xs:NMTOKEN` is the same as of type `xs:Name` with the only exception that it may not contain a colon.

### Argument

---

**\$value**

item value

### Examples

---

- The following queries generate valid values of type `xs:NMTOKEN`

```
xs:NMTOKEN(" -header")
```

```
xs:NMTOKEN(":R2D2")
```

- The following queries generate invalid values:

```
xs:NMTOKEN("+header")
```

The plus sign is not allowed in a name token.

```
xs:NMTOKEN("R2D2 C3P0")
```

It is interpreted as two tokens and thus cannot be a value of a single name token.

## xs:negativeInteger

Construct a `negativeInteger` value from an item value.

### Syntax

---

```
xs:negativeInteger(item $value) => negativeInteger
```

### Description

---

This is a constructor function that takes an item value as argument and returns a value of type `xs:negativeInteger`. If the argument is a literal, then the literal must be a valid lexical form of its type.

As specified in the respective section of the W3C recommendation [XML Schema Part 2: Datatypes](#), a value of type `xs:negativeInteger` represents an integer value less than zero. It is derived from the type `xs:nonPositiveInteger`. Please refer to the section *Derived built-in data types* in the *Tamino XML Schema User Guide* for information about the supported range of values.

### Argument

---

**\$value**

item value

### Examples

---

- This query creates a valid value of type `xs:negativeInteger`:

```
xs:negativeInteger(-234)
```

- This query does not create a valid value of type `xs:negativeInteger`:

```
xs:negativeInteger(0)
```

## xs:nonNegativeInteger

Construct a `nonNegativeInteger` value from an item value.

### Syntax

---

```
xs:nonNegativeInteger(item $value) => nonNegativeInteger
```

### Description

---

This is a constructor function that takes an item value as argument and returns a value of type `xs:nonNegativeInteger`. If the argument is a literal, then the literal must be a valid lexical form of its type.

As specified in the respective section of the W3C recommendation [XML Schema Part 2: Datatypes](#), a value of type `xs:negativeInteger` represents a negative integer value greater than or equal to zero. It is derived from the type `xs:integer`. Please refer to the section *Derived built-in data types* in the *Tamino XML Schema User Guide* for information about the supported range of values.

### Argument

---

**\$value**  
item value

### Examples

---

- This query creates a valid value of type `xs:nonNegativeInteger`:

```
xs:nonNegativeInteger(234)
```

- This query creates an invalid value:



```
xs:nonNegativeInteger(-7)
```

## xs:nonPositiveInteger

Construct a `nonPositiveInteger` value from an item value.

### Syntax

---

```
xs:nonPositiveInteger(item $value) => nonPositiveInteger
```

### Description

---

This is a constructor function that takes an item value as argument and returns a value of type `xs:nonPositiveInteger`. If the argument is a literal, then the literal must be a valid lexical form of its type.

As specified in the respective section of the W3C recommendation [XML Schema Part 2: Datatypes](#), a value of type `xs:positiveInteger` represents a negative integer value less than or equal to zero. It is derived from the type `xs:integer`. Please refer to the section *Derived built-in data types* in the *Tamino XML Schema User Guide* for information about the supported range of values.

### Argument

---

**\$value**  
item value

### Examples

---

- This query creates a valid value of type `xs:nonNegativeInteger`:

```
xs:nonPositiveInteger(-234)
```

- This query does not create a valid value of type `xs:nonNegativeInteger`:

```
xs:nonPositiveInteger(7)
```

## xs:normalizedString

Construct a `normalizedString` value from an item value.

### Syntax

---

```
xs:normalizedString(item $value) => normalizedString
```

### Description

---

This is a constructor function that takes an item value as argument and returns a value of type `xs:normalizedString`. If the argument is a literal, then the literal must be a valid lexical form of its type.

As specified in the respective section of the W3C recommendation [XML Schema Part 2: Datatypes](#), a value of type `xs:normalizedString` represents a white space normalized string value. It is derived from the type `xs:string` and the lexical form of `xs:normalizedString` is the set of all strings that do not contain the carriage return, line feed and tab characters.

### Argument

---

**\$value**  
item value

### Example

---

- This query creates a valid value of type `xs:normalizedString` that has only one space character between "deep" and "space":

```
xs:normalizedString("deep  
space  ")
```

## xs:positiveInteger

Construct a `positiveInteger` value from an item value.

### Syntax

```
xs:positiveInteger(item $value) => positiveInteger
```

### Description

This is a constructor function that takes an item value as argument and returns a value of type `xs:positiveInteger`. If the argument is a literal, then the literal must be a valid lexical form of its type.

As specified in the respective section of the W3C recommendation [XML Schema Part 2: Datatypes](#), a value of type `xs:positiveInteger` represents an integer value greater than zero. It is derived from the type `xs:nonNegativeInteger`. Please refer to the section *Derived built-in data types* in the *Tamino XML Schema User Guide* for information about the supported range of values.

### Argument

**\$value**

item value

### Examples

- This query creates a valid value of type `xs:positiveInteger`:

```
xs:positiveInteger(5)
```

- This query does not create a valid value of type `xs:positiveInteger`:

```
xs:positiveInteger(-234)
```

## xs:short

Construct a `short` value from an item value.

### Syntax

```
xs:short(item $value) => short
```

### Description

This is a constructor function that takes an item value as argument and returns a value of type `xs:short`. If the argument is a literal, then the literal must be a valid lexical form of its type.

As specified in the respective section of the W3C recommendation [XML Schema Part 2: Datatypes](#), `xs:short` is a subtype derived from `xs:int`. The lexical representation is the set of numerical values in the range from -32768 to 32767. If no sign is used, "+" is assumed.

### Argument

**\$value**

item value

### Examples

- The following queries construct valid values of type `xs:short`:

```
xs:short(44)
```

```
xs:short(-23)
```

```
xs:short(-0)
```

- The following queries do not construct valid values of type `xs:short`:

```
xs:short(34400)
```

```
xs:short(-+0)
```



## xs:string

Construct an `string` value from an item value.

### Syntax

```
xs:string(item $value) => string
```

### Description

This is a constructor function that takes an item value as argument and returns a value of type `xs:string`. If the argument is a literal, then the literal must be a valid lexical form of its type.

As specified in the respective section of the W3C recommendation [XML Schema Part 2: Datatypes](#), a value of type `xs:string` represents a set of finite-length sequences of characters that match the production of [Char](#), as defined in the [XML 1.0](#) recommendation. Every character has a corresponding Universal Character Set code point.

### Argument

**\$value**  
item value

### Example

- This query constructs a valid string value:

```
xs:string("!#$%&'()*+/,;@^_")
```

## xs:time

Construct a `time` value from an item value.

### Syntax

---

```
xs:time(item $value) => time
```

### Description

---

This is a constructor function that takes an item value as argument and returns a value of type `xs:time`. If the argument is a literal, then the literal must be a valid lexical form of its type.

As specified in the respective section of the W3C recommendation [XML Schema Part 2: Datatypes](#), a value of type `xs:time` represents a specific instant of time by a combination of date and time of day values. The lexical form is `hh:mm:ss` where "hh" represents the hour, "mm" the minute, "ss" the second. Fractions of a second are appended with a leading ".". A timezone can be appended. You may not truncate digits on the left side.

### Argument

---

**\$value**

item value

### Example

---

- This query constructs a valid time value:

```
xs:time("12:20:46.275+01:00")
```

This represents 12:20:46 and 275/1000 seconds CET. The timezone CET is 1 hour after Coordinated Universal Time (UTC).

## xs:token

Construct a `token` value from an item value.

### Syntax

```
xs:token(item $value) => token
```

### Description

This is a constructor function that takes an item value as argument and returns a value of type `xs:token`. If the argument is a literal, then the literal must be a valid lexical form of its type.

As specified in the respective section of the W3C recommendation [XML Schema Part 2: Datatypes](#), a value of type `xs:token` represents an XML token value. It is derived from the type `xs:normalizedString` and the lexical form of `xs:token` is the set of all white space normalized strings that do not contain the carriage return, line feed and tab characters.

### Argument

**\$value**  
item value

### Example

- This query creates a valid value of type `xs:normalizedString` that has only one space character between "deep" and "space":

```
xs:token("deep  
space  ")
```

## xs:unsignedByte

Construct an `unsignedByte` value from an item value.

### Syntax

---

```
xs:unsignedByte(item $value) => unsignedByte
```

### Description

---

This is a constructor function that takes an item value as argument and returns a value of type `xs:unsignedByte`. If the argument is a literal, then the literal must be a valid lexical form of its type.

As specified in the respective section of the W3C recommendation [XML Schema Part 2: Datatypes](#), `xs:unsignedByte` is a subtype derived from `xs:unsignedShort`. The lexical representation is the set of unsigned numerical values in the range from 0 to 255.

### Argument

---

**\$value**  
item value

### Examples

---

- The following queries construct valid values of type `xs:unsignedByte`:

```
xs:unsignedByte(44)
```

```
xs:unsignedByte(-0)
```

- The following queries do not construct valid values of type `xs:unsignedByte`:

```
xs:unsignedByte(4400)
```

```
xs:unsignedByte(-223)
```

## xs:unsignedInt

Construct an `unsignedInt` value from an item value.

### Syntax

---

```
xs:unsignedInt(item $value) => unsignedInt
```

### Description

---

This is a constructor function that takes an item value as argument and returns a value of type `xs:unsignedInt`. If the argument is a literal, then the literal must be a valid lexical form of its type.

As specified in the respective section of the W3C recommendation [XML Schema Part 2: Datatypes](#), a value of type `xs:unsignedInt` represents an integer value derived from `xs:unsignedLong`. The lexical representation is the set of unsigned numerical values in the range from 0 to 4294967295.

### Argument

---

**\$value**

item value

### Examples

---

- The following queries construct valid values of type `xs:unsignedInt`:

```
xs:unsignedInt(4444)
```

```
xs:unsignedInt(-0)
```

- The following queries do not construct valid values of type `xs:unsignedInt`:

```
xs:unsignedInt(4294967296)
```

```
xs:unsignedInt(-223)
```

## xs:unsignedLong

Construct an `unsignedLong` value from an item value.

### Syntax

---

```
xs:unsignedLong(item $value) => unsignedLong
```

### Description

---

This is a constructor function that takes an item value as argument and returns a value of type `xs:unsignedLong`. If the argument is a literal, then the literal must be a valid lexical form of its type.

As specified in the respective section of the W3C recommendation [XML Schema Part 2: Datatypes](#), a value of type `xs:unsignedLong` represents an integer value derived from `xs:nonNegativeInteger`. The lexical representation is the set of unsigned numerical values in the range from 0 to 18446744073709551615.

### Argument

---

**\$value**  
item value

### Example

---

- The following queries construct valid values of type `xs:unsignedLong`:

```
xs:unsignedLong(18446744073709551615)
```

```
xs:unsignedLong(-0)
```

- The following queries do not construct valid values of type `xs:unsignedInt`:



```
xs:unsignedLong(18446744073709551616)
```

```
xs:unsignedLong(-223)
```

## xs:unsignedShort

Construct an `unsignedShort` value from an item value.

### Syntax

---

```
xs:unsignedShort(item $value) => unsignedShort
```

### Description

---

This is a constructor function that takes an item value as argument and returns a value of type `xs:unsignedShort`. If the argument is a literal, then the literal must be a valid lexical form of its type.

As specified in the respective section of the W3C recommendation [XML Schema Part 2: Datatypes](#), a value of type `xs:unsignedShort` represents an integer value derived from `xs:unsignedInt`. The lexical representation is the set of unsigned numerical values in the range from 0 to 65535.

### Argument

---

**\$value**  
item value

### Example

---

- The following queries construct valid values of type `xs:unsignedShort`:

```
xs:unsignedShort(65535)
```

```
xs:unsignedShort(-0)
```

- The following queries do not construct valid values of type `xs:unsignedInt`:

```
xs:unsignedShort(65536)
```

```
xs:unsignedShort(-223)
```



# Index

---

## A

AbbreviatedStep, 26  
AbsolutePathExpr, 28  
AdditiveExpr, 29  
AndExpr, 30  
AnyKindTest, 31  
AttributeList, 32  
AttributeTest, 33  
AttributeValue, 34  
AttributeValueContent, 35  
Axis, 37  
AxisStep, 39

## C

CommentTest, 40  
CompCommentConstructor, 41  
CompExpr, 42  
CompPIConstructor, 43  
CompTextConstructor, 44  
ComputedAttributeConstructor, 45  
ComputedDocumentConstructor, 47  
ComputedElementConstructor, 49  
Constructor, 51

## D

DefaultCollationDecl, 52  
DefaultNamespaceDecl, 54  
DeleteClause, 56  
DirectCommentConstructor, 57

## E

ElementConstructor, 58  
ElementContent, 60  
ElementNameOrFunctionCall, 62  
ElementTest, 63  
EnclosedExpr, 64  
Expr, 65  
ExprSequence, 67

## F

FLWORExpr, 68  
FLWUExpr, 71  
fn:abs, 183

fn:avg, 184  
fn:boolean, 185  
fn:ceiling, 186  
fn:collection, 188  
fn:compare, 189  
fn:concat, 191  
fn:contains, 192  
fn:count, 194  
fn:current-date, 195  
fn:current-dateTime, 196  
fn:current-time, 197  
fn:data, 198  
fn:day-from-date, 199  
fn:day-from-dateTime, 200  
fn:deep-equal, 201  
fn:distinct-values, 204  
fn:ends-with, 206  
fn:expanded-QName, 208  
fn:false, 209  
fn:floor, 210  
fn:get-local-name-from-QName, 212  
fn:get-namespace-from-QName, 213  
fn:hours-from-dateTime, 214  
fn:hours-from-time, 215  
fn:id, 216  
fn:idref, 217  
fn:last, 218  
fn:local-name, 219  
fn:lower-case, 220  
fn:matches, 221  
fn:max, 223  
fn:min, 224  
fn:minutes-from-dateTime, 225  
fn:minutes-from-time, 226  
fn:month-from-date, 227  
fn:month-from-dateTime, 228  
fn:namespace-uri, 229  
fn:node-name, 230  
fn:normalize-space, 231  
fn:not, 232  
fn:position, 233  
fn:put, 234  
fn:replace, 236  
fn:reverse, 238  
fn:root, 239  
fn:round, 240  
fn:seconds-from-dateTime, 242  
fn:seconds-from-time, 243  
fn:starts-with, 244  
fn:string, 245

fn:string-join, 247  
fn:string-length, 249  
fn:subsequence, 250  
fn:substring, 252  
fn:substring-after, 254  
fn:substring-before, 256  
fn:sum, 258  
fn:tokenize, 259  
fn:true, 260  
fn:upper-case, 261  
fn:year-from-date, 262  
fn:year-from-dateTime, 263  
ForClause, 73  
ft:proximity-contains, 264  
ft:text-contains, 267  
FunctionDecl, 74

## G

GeneralComp, 76  
GeneralStep, 78

## I

IfExpr, 79  
InsertClause, 81  
IntersectExceptExpr, 83  
ItemType, 84

## K

KindTest, 85

## L

LetClause, 86  
LibraryModule, 87  
Literal, 88

## M

MainModule, 89  
ModuleDecl, 90  
ModuleImport, 91  
MultiplicativeExpr, 93

## N

NamespaceDecl, 95  
NameTest, 97  
NoAxisStep, 98  
NodeComp, 99  
NodeTest, 101  
NumericLiteral, 102

## O

OrderByClause, 103  
OrExpr, 105

## P

ParenthesizedExpr, 106

PathExpr, 107  
Pragma, 108  
PrimaryExpr, 109  
ProcessingInstructionTest, 111  
Prolog, 112

## Q

Query Execution Plan, 134

## R

RangeExpr, 114  
RelativePathExpr, 115  
RenameClause, 117  
ReplaceClause, 118

## S

SequenceType, 119  
SerializationSpec, 120  
SortExpr, 125  
SortSpecList, 127  
StepExpr, 129  
StepQualifiers, 130

## T

TaminoQPIExecution, 132  
TaminoQPIExplain, 134  
TaminoQPIInline, 136  
TaminoQPIOptimization, 137  
tdf:getProperties, 269  
tdf:getProperty, 270  
tdf:isDescendantOf, 271  
tdf:mkcol, 273  
tdf:resource, 274  
tdf:setProperty, 275  
TextTest, 138  
tf:broaderTerm, 276  
tf:broaderTerms, 278  
tf:containsAdjacentText, 280  
tf:containsNearText, 282  
tf:containsText, 284  
tf:content-type, 286  
tf:createAdjacentTextReference, 287  
tf:createNearTextReference, 289  
tf:createNodeReference, 291  
tf:createTextNode, 292  
tf:createTextReference, 294  
tf:document, 297  
tf:get-current-user, 301  
tf:getCollation, 298  
tf:getCollection, 300  
tf:getDocname, 302  
tf:getInId, 303  
tf:getLastModified, 304  
tf:highlight, 306  
tf:narrowerTerm, 308  
tf:narrowerTerms, 310  
tf:nonXML-kind, 312  
tf:parse, 313  
tf:phonetic, 314  
tf:query, 316

tf:serialize, 317  
tf:setDocname, 318  
tf:stem, 319  
tf:synonym, 321  
tf:text-content, 323  
TreatExpr, 139

## U

UnaryExpr, 140  
UnionExpr, 142  
UpdateExpr, 143  
UpdateIfExpr, 144  
UpdateSequence, 146  
UpdatingFunction, 147  
UpdatingFunctionDecl, 148

## V

ValueComp, 149  
VarDecl, 151  
VersionDecl, 153

## W

WhereClause, 154  
Wildcard, 155

## X

XmlProcessingInstruction, 157  
XQueryModule  
    syntax construct, 158  
xs:anyURI, 324  
xs:base64Binary, 325  
xs:boolean, 326  
xs:byte, 328  
xs:date, 330  
xs:dateTime, 332  
xs:decimal, 334  
xs:double, 335  
xs:duration, 336  
xs:ENTITY, 338  
xs:float, 340  
xs:gDay, 341  
xs:gMonth, 342  
xs:gMonthDay, 343  
xs:gYear, 344  
xs:gYearMonth, 346  
xs:hexBinary, 348  
xs:ID, 349  
xs:IDREF, 351  
xs:int, 353  
xs:integer, 354  
xs:language, 355  
xs:long, 356  
xs:Name, 358  
xs:NCName, 360  
xs:negativeInteger, 364  
xs:NMTOKEN, 362  
xs:nonNegativeInteger, 366  
xs:nonPositiveInteger, 368  
xs:normalizedString, 370  
xs:positiveInteger, 371

xs:short, 373  
xs:string, 375  
xs:time, 376  
xs:token, 377  
xs:unsignedByte, 378  
xs:unsignedInt, 380  
xs:unsignedLong, 382  
xs:unsignedShort, 384

---