



## XQuery Reference





**XQuery Reference**

**Note**

Note: Before using this information and the product it supports, read the information in Notices.

**Edition Notice**

This document contains proprietary information of IBM. It is provided under a license agreement and is protected by copyright law. The information contained in this publication does not include any product warranties, and any statements provided in this manual should not be interpreted as such.

You can order IBM publications online or through your local IBM representative.

- To order publications online, go to the IBM Publications Center at [www.ibm.com/shop/publications/order](http://www.ibm.com/shop/publications/order).
- To find your local IBM representative, go to the IBM Directory of Worldwide Contacts at [www.ibm.com/planetwide](http://www.ibm.com/planetwide).

To order DB2 publications from DB2 Marketing and Sales in the United States or Canada, call 1-800-IBM-4YOU (426-4968).

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 2006. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

---

# Contents

Tables . . . . .	ix
Figures . . . . .	xi
<b>Chapter 1. DB2 XQuery concepts . . . . .</b>	<b>1</b>
Introduction to XQuery . . . . .	1
Comparison of XQuery to SQL . . . . .	2
Retrieving DB2 data with XQuery functions . . . . .	2
XQuery and XPath data model . . . . .	4
Sequences and items . . . . .	4
Atomic values . . . . .	4
Node hierarchies . . . . .	5
Node properties . . . . .	7
Node kinds . . . . .	7
Document order of nodes . . . . .	10
Node identity . . . . .	10
Typed values and string values of nodes . . . . .	10
Serialization of the XDM . . . . .	11
XML namespaces and QNames . . . . .	12
Qualified names (QNames) . . . . .	12
Statically known namespaces . . . . .	13
Language conventions . . . . .	14
Case sensitivity . . . . .	14
Whitespace . . . . .	14
Comments . . . . .	15
Where to find more information about XQuery . . . . .	15
<b>Chapter 2. Type system . . . . .</b>	<b>17</b>
The type hierarchy . . . . .	17
Types by category . . . . .	18
Constructor functions for built-in data types . . . . .	22
Type casting . . . . .	23
anyAtomicType data type . . . . .	25
anySimpleType data type . . . . .	25
anyType data type . . . . .	26
anyURI data type . . . . .	26
base64Binary data type . . . . .	26
boolean data type . . . . .	26
byte data type . . . . .	26
date data type . . . . .	26
dateTime data type . . . . .	27
dayTimeDuration data type . . . . .	28
decimal data type . . . . .	29
double data type . . . . .	29
duration data type . . . . .	30
ENTITY data type . . . . .	31
float data type . . . . .	31
gDay data type . . . . .	31
gMonth data type . . . . .	32
gMonthDay data type . . . . .	32
gYear data type . . . . .	32
gYearMonth data type . . . . .	33
hexBinary data type . . . . .	33

ID data type	33
IDREF data type	33
int data type	34
integer data type	34
language data type	34
long data type	34
Name data type	34
NCName data type	34
negativeInteger data type	34
NMTOKEN data type	35
nonNegativeInteger data type	35
nonPositiveInteger data type	35
normalizedString data type	35
NOTATION data type	35
positiveInteger data type	35
QName data type	35
short data type	36
string data type	36
time data type	36
token data type	37
unsignedByte data type	37
unsignedInt data type	37
unsignedLong data type	37
unsignedShort data type	37
untyped data type	37
untypedAtomic data type	37
yearMonthDuration data type	38
<b>Chapter 3. Prolog</b>	<b>39</b>
Version declaration	39
Boundary-space declaration	40
Construction declaration	40
Copy-namespaces declaration	41
Default element/type namespace declaration	42
Default function namespace declaration	42
Empty order declaration	43
Ordering mode declaration	44
Namespace declaration	44
<b>Chapter 4. Expressions</b>	<b>47</b>
Concepts for expression processing	47
Dynamic context and focus	47
Precedence	47
Order of results in XQuery expressions	48
Atomization	50
Subtype substitution	50
Type promotion	51
Effective Boolean value	51
Primary expressions	52
Literals	52
Variable references	54
Parenthesized expression	55
Context item expressions	55
Function calls	55
Path expressions	56
Syntax of path expressions	57

Axis steps . . . . .	58
Abbreviated syntax for path expressions . . . . .	61
Predicates . . . . .	63
Sequence expressions . . . . .	64
Expressions that construct sequences . . . . .	64
Filter expressions . . . . .	65
Expressions for combining sequences of nodes . . . . .	66
Arithmetic expressions . . . . .	67
Comparison expressions . . . . .	69
Value comparisons . . . . .	69
General comparisons . . . . .	71
Node comparisons . . . . .	73
Logical expressions. . . . .	73
Constructors . . . . .	75
Enclosed expressions in constructors . . . . .	75
Direct element constructors . . . . .	76
Computed element constructors . . . . .	83
Computed attribute constructors . . . . .	84
Document node constructors . . . . .	85
Text node constructors . . . . .	86
Processing instruction constructors . . . . .	86
Comment constructors . . . . .	88
FLWOR expressions . . . . .	89
Syntax of FLWOR expressions . . . . .	89
<b>for</b> and <b>let</b> clauses . . . . .	90
<b>where</b> clauses . . . . .	94
<b>order by</b> clauses . . . . .	95
<b>return</b> clauses . . . . .	97
FLWOR examples . . . . .	97
Conditional expressions. . . . .	100
Quantified expressions . . . . .	101
Cast expressions . . . . .	102
<b>Chapter 5. Built-in functions . . . . .</b>	<b>105</b>
Functions by category . . . . .	105
abs function . . . . .	109
avg function . . . . .	110
boolean function . . . . .	111
ceiling function . . . . .	111
codepoints-to-string function . . . . .	112
compare function . . . . .	113
concat function . . . . .	114
contains function . . . . .	114
count function . . . . .	115
current-date function . . . . .	115
current-dateTime function . . . . .	116
current-time function . . . . .	116
data function . . . . .	117
dateTime function . . . . .	117
deep-equal function . . . . .	118
default-collation function . . . . .	119
distinct-values function . . . . .	120
empty function . . . . .	121
ends-with function . . . . .	121
exactly-one function . . . . .	122
exists function . . . . .	122

false function	123
floor function	123
implicit-timezone function	124
in-scope-prefixes function	124
index-of function	125
insert-before function	126
last function	126
local-name function	127
local-name-from-QName function	128
lower-case function	128
matches function	129
max function	130
min function	131
name function	132
namespace-uri function	133
namespace-uri-for-prefix function	134
namespace-uri-from-QName function	134
node-name function	135
normalize-space function	135
normalize-unicode function	136
not function	137
number function	137
one-or-more function	138
position function	138
QName function	139
remove function	140
replace function	140
resolve-QName function	142
reverse function	143
root function	143
round function	144
round-half-to-even function	145
sqlquery function	146
starts-with function	147
string function	148
string-join function	148
string-length function	149
string-to-codepoints function	149
subsequence function	150
substring function	151
substring-after function	151
substring-before function	152
sum function	153
tokenize function	154
translate function	155
true function	156
unordered function	157
upper-case function	157
xmlcolumn function	158
zero-or-one function	159

<b>Chapter 6. Limits</b>	161
Limits for XQuery data types	161
Size limits	162

<b>DB2 technical library in PDF format</b>	163
--	-----



<b>Ordering printed DB2 books</b> . . . . .	165
<b>DB2 troubleshooting information</b> . . . . .	167
<b>Notices</b> . . . . .	169
<b>Contacting IBM</b> . . . . .	173
<b>Index</b> . . . . .	175



---

## Tables

1. String values and typed values of nodes . . . . .	10
2. Predeclared namespaces in DB2 XQuery . . . . .	13
3. Generic data types . . . . .	19
4. Untyped data types . . . . .	19
5. String data types . . . . .	19
6. Numeric data types . . . . .	20
7. Date, time, and duration data types . . . . .	20
8. Other data types . . . . .	21
9. Primitive type casting, part 1 (targets from xdt:untypedAtomic to xs:dateTime) . . . . .	23
10. Primitive type casting, part 2 (targets from xs:time to xs:NOTATION) . . . . .	24
11. Predeclared namespaces in DB2 XQuery . . . . .	45
12. Precedence in DB2 XQuery . . . . .	47
13. Summary of ordering of results in XQuery expressions . . . . .	49
14. EBVs returned for specific types of values in XQuery . . . . .	52
15. Predefined entity references in DB2 XQuery . . . . .	53
16. Supported axes in DB2 XQuery . . . . .	59
17. Supported name tests in DB2 XQuery . . . . .	60
18. Supported kind tests in DB2 XQuery . . . . .	60
19. Abbreviated syntax for path expressions . . . . .	61
20. Unabbreviated syntax and abbreviated syntax . . . . .	62
21. XQuery operators for combining sequences of nodes . . . . .	66
22. Arithmetic operators in XQuery. . . . .	67
23. Valid types for operands of arithmetic expressions . . . . .	68
24. Value comparison operators in XQuery. . . . .	69
25. Value comparison operators in XQuery. . . . .	71
26. Node comparison operators in XQuery . . . . .	73
27. Logical expression operators in XQuery . . . . .	73
28. Results of logical expressions based on EBVs of operands . . . . .	74
29. Representation of special characters in attribute values . . . . .	77
30. Representation of special characters in element content . . . . .	78
31. Comparison of <b>for</b> and <b>let</b> clauses in FLWOR expressions . . . . .	93
32. EBVs returned for specific types of values in XQuery . . . . .	111
33. Deep equality for nodes in a sequence . . . . .	119
34. Limits for XQuery numeric data types . . . . .	161
35. Limits for XQuery date, time, and duration data types . . . . .	162
36. DB2 technical information . . . . .	163
37. Technical information specific to DB2 Connect . . . . .	164
38. WebSphere Information Integration technical information. . . . .	164



---

## Figures

1.	Structure of a typical query in XQuery . . . . .	1
2.	Data model diagram for products.xml document . . . . .	6
3.	DB2 XQuery type hierarchy . . . . .	18



# Chapter 1. DB2 XQuery concepts

The following topics introduce basic XQuery concepts and describe how XQuery works with a DB2® database.

## Introduction to XQuery

XQuery is a functional programming language that was designed by the World Wide Web Consortium (W3C) to meet specific requirements for querying XML data.

Unlike relational data, which is predictable and has a regular structure, XML data is highly variable. XML data is often unpredictable, sparse, and self-describing.

Because the structure of XML data is unpredictable, the queries that you need to perform on XML data often differ from typical relational queries. The XQuery language provides the flexibility required to perform these kinds of operations. For example, you might need to create XML queries that perform the following operations:

- Search XML data for objects that are at unknown levels of the hierarchy.
- Perform structural transformations on the data (for example, you might want to invert a hierarchy).
- Return results that have mixed types.

In XQuery, expressions are the main building blocks of a query. Expressions can be nested and form the body of a query. A query can also have a prolog before this body. The *prolog* contains a series of declarations that define the processing environment for the query. The *query body* consists of an expression that defines the result of the query. This expression can be composed of multiple XQuery expressions that are combined using operators or keywords.

Figure 1 illustrates the structure of a typical query. In this example, the prolog contains two declarations: a version declaration, which specifies the version of the XQuery syntax to use to process the query, and a default namespace declaration that specifies the namespace URI to use for unprefixed element and type names. The query body contains an expression that constructs a `price_list` element. The content of the `price_list` element is a list of product elements that are sorted in descending order by price.



Figure 1. Structure of a typical query in XQuery

### Related concepts

---

## Comparison of XQuery to SQL

DB2 supports storing well-formed XML data in a column of a table and retrieving the XML data from the database by using SQL, XQuery, or a combination of SQL and XQuery. Both languages are supported as primary query languages, and both languages provide functions for invoking the other language.

### XQuery

A query that invokes XQuery directly begins with the keyword XQUERY. This keyword indicates that XQuery is being used and that the DB2 server must therefore use case sensitivity rules that apply to the XQuery language. Error handling is based on the interfaces that are used to process XQuery expressions. XQuery errors are reported with an SQLCODE and SQLSTATE in the same way that SQL error errors are reported. No warnings are returned from processing XQuery expressions. XQuery obtains data by calling functions that extract XML data from DB2 tables and views. XQuery can also be invoked from an SQL query. In this case, the SQL query can pass XML data to XQuery in the form of bound variables. XQuery supports various expressions for processing XML data and for constructing new XML objects such as elements and attributes. The programming interface to XQuery provides facilities similar to those of SQL to prepare queries and retrieve query results.

**SQL** SQL provides capabilities to define and instantiate values of the XML data type. Strings that contain well-formed XML documents can be parsed into XML values, optionally validated against an XML schema, and inserted or updated in tables. Alternatively, XML values can be constructed by using SQL constructor functions, which convert other relational data into XML values. Functions are also provided to query XML data by using XQuery and to convert XML data into a relational table for use within an SQL query. Data can be cast between SQL and XML data types in addition to serializing XML values into string data.

SQL/XML provides the following functions and predicates for calling XQuery from SQL:

### XMLQUERY

XMLQUERY is a scalar function that takes an XQuery expression as an argument and returns an XML sequence. The function includes optional parameters that can be used to pass SQL values to the XQuery expression as XQuery variables. The XML values that are returned by XMLQUERY can be further processed within the context of the SQL query.

### XMLTABLE

XMLTABLE is a table function that uses XQuery expressions to generate an SQL table from XML data, which can be further processed by SQL.

### XMLEXISTS

XMLEXISTS is an SQL predicate that determines if an XQuery expression returns a sequence of one or more items (and not an empty sequence).

### Related concepts

"Retrieving DB2 data with XQuery functions"

---

## Retrieving DB2 data with XQuery functions

In XQuery, a query can call one of the following functions to obtain input XML data from a DB2 database: db2-fn:sqlquery and db2-fn:xmlcolumn.

The function db2-fn:xmlcolumn retrieves an entire XML column, whereas db2-fn:sqlquery retrieves XML values that are based on an SQL fullselect.

### db2-fn:xmlcolumn

The db2-fn:xmlcolumn function takes a string literal argument that identifies an XML column in a table or a view and returns a sequence of XML values that are in that column. The argument of



this function is case sensitive. The string literal argument must be a qualified column name of type XML. This function allows you to extract a whole column of XML data without applying a search condition.

In the following example, the query uses the `db2-fn:xmlcolumn` function to get all of the purchase orders in the `PURCHASE_ORDER` column of the `BUSINESS.ORDERS` table. The query then operates on this input data to extract the cities from the shipping address in these purchase orders. The result of the query is a list of all cities to which orders are shipped:

```
db2-fn:xmlcolumn('BUSINESS.ORDERS.PURCHASE_ORDER')/shipping_address/city
```

### **db2-fn:sqlquery**

The `db2-fn:sqlquery` function takes a string argument that represents a fullselect and returns an XML sequence that is a concatenation of the XML values that are returned by the fullselect. The fullselect must specify a single-column result set, and the column must have a data type of XML. Specifying a fullselect allows you to use the power of SQL to present XML data to XQuery.

In the following example, a table called `BUSINESS.ORDERS` contains an XML column called `PURCHASE_ORDER`. The query in the example uses the `db2-fn:sqlquery` function to call SQL to get all of the purchase orders where the ship date is June 15, 2005. The query then operates on this input data to extract the cities from the shipping addresses in these purchase orders. The result of the query is a list of all of the cities to which orders are shipped on June 15:

```
db2-fn:sqlquery("
SELECT purchase_order FROM business.orders
WHERE ship_date = '2005-06-15' ")/shipping_address/city
```

**Important:** An XML sequence that is returned by the `db2-fn:sqlquery` or `db2-fn:xmlcolumn` function can contain any XML values, including atomic values and nodes. These functions do not always return a sequence of well-formed documents. For example, the function might return a single atomic value, like 36, as an instance of the XML data type.

SQL and XQuery have different conventions for case-sensitivity of names. You should be aware of these differences when using the `db2-fn:sqlquery` and `db2-fn:xmlcolumn` functions.

### **SQL is not a case-sensitive language**

By default, all ordinary identifiers, which are used in SQL statements, are automatically converted to uppercase. Therefore, the names of SQL tables and columns are customarily uppercase names, such as `BUSINESS.ORDERS` and `PURCHASE_ORDER` in the previous examples. In an SQL statement, these columns can be referenced by using lowercase names, such as `business.orders` and `purchase_order`, which are automatically converted to uppercase during processing of the SQL statement. (You can also create a case-sensitive name that is called a *delimited identifier* in SQL by enclosing the name in double quotation marks.)

### **XQuery is a case-sensitive language**

XQuery does not convert lowercase names to uppercase. This difference can lead to some confusion when XQuery and SQL are used together. The string that is passed to `db2-fn:sqlquery` is interpreted as an SQL query and is parsed by the SQL parser, which converts all names to uppercase. Thus, in the `db2-fn:sqlquery` example, the table name `business.orders` and the column names `purchase_order` and `ship_date` can appear in either uppercase or lowercase. The operand of `db2-fn:xmlcolumn`, however, is not an SQL query. The operand is a case-sensitive XQuery string literal that represents the name of a column. Because the actual name of the column is `BUSINESS.ORDERS.PURCHASE_ORDER`, this name must be specified in uppercase in the operand of `db2-fn:xmlcolumn`.

#### **Related concepts**

“Comparison of XQuery to SQL” on page 2

#### **Related reference**

“xmlcolumn function” on page 158

“sqlquery function” on page 146

---

## XQuery and XPath data model

XQuery expressions operate on instances of the XQuery and XPath data model (XDM) and return instances of the data model. The XDM provides an abstract representation of one or more XML documents or fragments. The data model defines all permissible values of expressions in XQuery, including values that are used during intermediate calculations.

Parsing of XML data into the XDM and validating the data against a schema occur before data is processed by XQuery. During data model generation, the input XML document is parsed and converted into an instance of the XDM. The document can be parsed with or without validation.

The XDM is described in terms of sequences of atomic values and nodes.

### Related concepts

“Serialization of the XDM” on page 11

## Sequences and items

An instance of the XDM is a sequence. A *sequence* is an ordered collection of zero or more items. An *item* is either an atomic value or a node.

A sequence can contain nodes, atomic values, or any mixture of nodes and atomic values. For example, each of the following values is a sequence:

- 36
- <dog/>
- (2, 3, 4)
- (36, <dog/>, "cat")
- ()
- an XML document

**Note:** These examples use a notation to represent sequences that is consistent with the syntax that is used to construct sequences in XQuery. Each item in the sequence is separated by a comma. The entire sequence is enclosed in parentheses. A pair of empty parentheses represents an empty sequence. A single item that appears on its own is equivalent to a sequence that contains one item. For example, there is no distinction between the sequence (36) and the atomic value 36.

Sequences cannot be nested. When two sequences are combined, the result is always a flattened sequence of nodes and atomic values. For example, appending the sequence (2, 3) to the sequence (3, 5, 6) results in the single sequence (3, 5, 6, 2, 3). Combining these sequences does not produce the sequence (3, 5, 6, (2, 3)) because nested sequences never occur.

A sequence that contains zero items is called an *empty sequence*. Empty sequences can be used to represent missing or unknown information.

### Related reference

“Sequence expressions” on page 64

“Context item expressions” on page 55

“General comparisons” on page 71

## Atomic values

An *atomic value* is an instance of one of the built-in atomic data types that are defined by XML Schema. These data types include strings, integers, decimals, dates, and other atomic types. These types are described as atomic because they cannot be subdivided.

Unlike nodes, atomic values do not have an identity. Every instance of an atomic value (for example, the integer 7) is identical to every other instance of that value.

The following examples are some of ways that atomic values are made:

- Extracted from nodes through a process called atomization. Atomization is used by expressions whenever a sequence of atomic values is required.
- Specified as a numeric or string literal. Literals are interpreted by XQuery as atomic values. For example, the following literals are interpreted as atomic values:
  - "this is a string" (type is xs:string)
  - 45 (type is xs:integer)
  - 1.44 (type is xs:decimal)
- Computed by constructor functions. For example, the following constructor function builds a value of type xs:date out of the string "2005-01-01":

```
xs:date("2005-01-01")
```
- Returned by the built-in functions `fn:true()` and `fn:false()`. These functions return the boolean values `true` and `false`. These values cannot be expressed as literals.
- Returned by many kinds of expressions, such as arithmetic expressions and logical expressions.

#### **Related concepts**

"Atomization" on page 50

Chapter 2, "Type system," on page 17

Chapter 4, "Expressions," on page 47

"Constructor functions for built-in data types" on page 22

#### **Related reference**

"Literals" on page 52

"true function" on page 156

"false function" on page 123

## **Node hierarchies**

The nodes of a sequence form one or more *hierarchies*, or *trees*, that consist of a root node and all of the nodes that are reachable directly or indirectly from the root node. Every node belongs to exactly one hierarchy, and every hierarchy has exactly one root node. DB2 supports six node kinds: document, element, attribute, text, processing instruction, and comment.

The following XML document, `products.xml`, includes a root element, named `products`, which contains product elements. Each product element has an attribute named `pid` (product ID) and a child element named `description`. The description element contains child elements named `name` and `price`.

```
<products>
  <product xmlns="http://posample.org" pid="10">
    <description>
      <name>Fleece jacket</name>
      <price>19.99</price>
    </description>
  </product>
  <product xmlns="http://posample.org" pid="11">
    <description>
      <name>Nylon pants</name>
      <price>9.99</price>
    </description>
  </product>
</products>
```

Figure 2 shows a simplified representation of the data model for products.xml. The figure includes a document node (D), element nodes (E), attribute nodes (A), and text nodes (T).

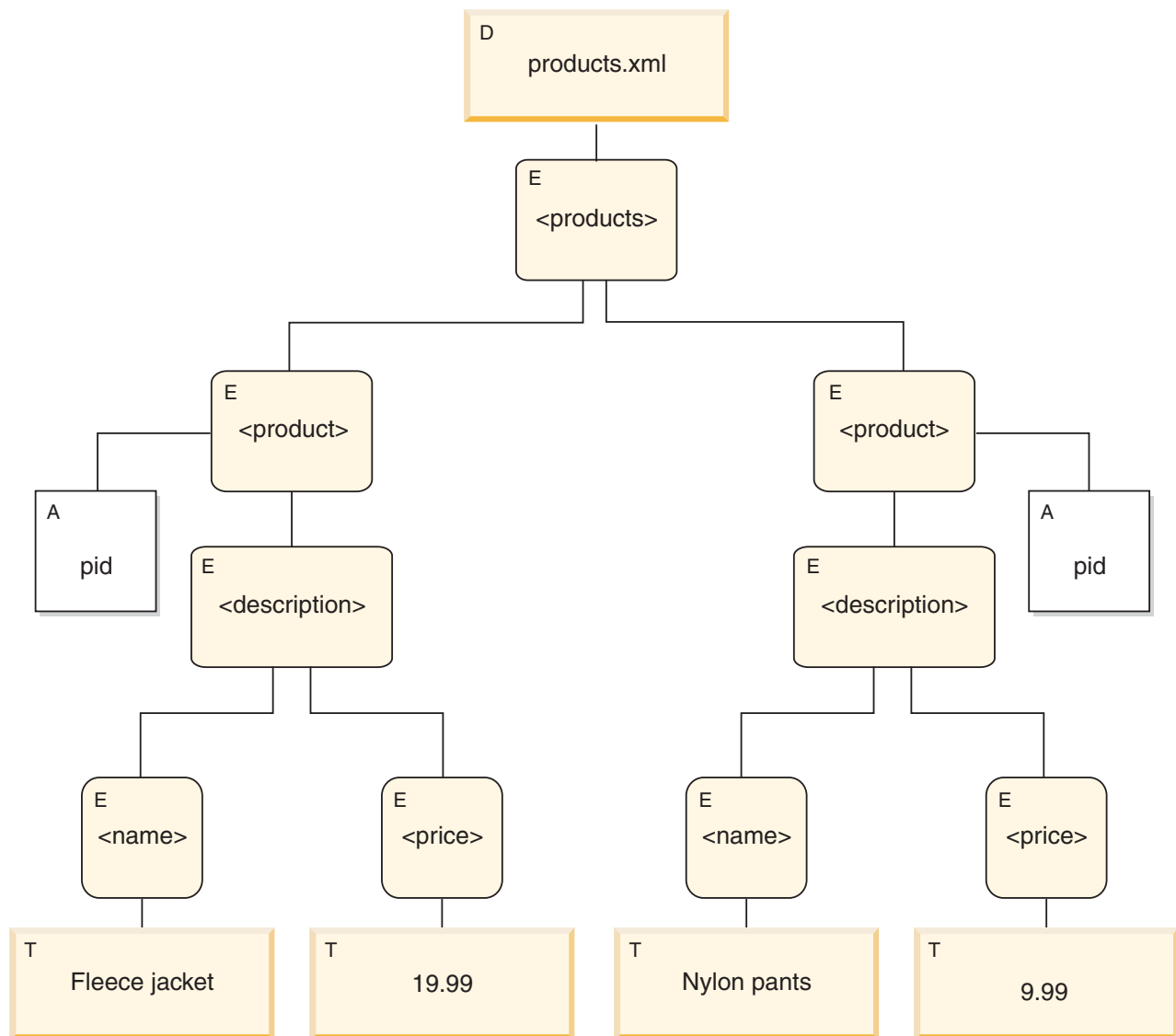


Figure 2. Data model diagram for products.xml document

As the example illustrates, a node can have other nodes as children, thus forming one or more *node hierarchies*. In the example, the element product is a child of products. The element description is a child of product. The elements name and price are children of the element description. The text node with the value Fleece Jacket is a child of the element name, and the text node 19.99 is a child of the element price.

**Related concepts**

- “Document order of nodes” on page 10
- “Node kinds” on page 7

**Related reference**

- “Path expressions” on page 56
- “Expressions for combining sequences of nodes” on page 66

## Node properties

Each node has *properties* that describe characteristics of that node. For example, a node's properties might include the name of the node, its children, its parent, its attributes, and other information that describes the node. The node kind determines which properties are present for specific nodes.

A node can have one or more of the following properties:

- **node-name.** The name of the node, expressed as a QName.
- **parent.** The node that is the parent of the current node.
- **type-name.** The dynamic (run-time) type of the node (also known as the *type annotation*).
- **children.** The sequence of nodes that are children of the current node.
- **attributes.** The set of attribute nodes that belong to the current node.
- **string-value.** A string value that can be extracted from the node.
- **typed-value.** A sequence of zero or more atomic values that can be extracted from the node.
- **in-scope namespaces.** The in-scope namespaces that are associated with the node.
- **content.** The content of the node.

### Related concepts

“Node kinds”

“Typed values and string values of nodes” on page 10

## Node kinds

DB2 supports six node kinds: document, element, attribute, text, processing instruction, and comment.

### Related concepts

“Node hierarchies” on page 5

### Related reference

“Constructors” on page 75

## Document nodes

A document node encapsulates an XML document.

A document node can have zero or more children. The children can include element nodes, processing instruction nodes, comment nodes, and text nodes.

The string value of a document node is equal to the concatenated contents of all its descendant text nodes in document order. The type of the string value is `xs:string`. The typed value of a document node is the same as its string value, except that the type of the typed value is `xdt:untypedAtomic`.

A document node has the following node properties:

- children, possibly empty
- string-value
- typed-value

Document nodes can be constructed in XQuery expressions by using computed constructors. A sequence of document nodes can also be returned by the `db2-fn:xmlcolumn` function.

### Related concepts

“Node properties”

### Related reference

“Document node constructors” on page 85

“xmlcolumn function” on page 158

## Element nodes

An element node encapsulates an XML element.

An element can have zero or one parent, and zero or more children. The children can include element nodes, processing instruction nodes, comment nodes, and text nodes. Document and attribute nodes are never children of element nodes. However, an element node is considered to be the parent of its attributes. The attributes of an element node must have unique QNames.

An element node has the following node properties:

- node-name
- parent, possibly empty
- type-name
- children, possibly empty
- attributes, possibly empty
- string-value
- typed-value
- in-scope-namespaces

Element nodes can be constructed in XQuery expressions by using direct or computed constructors.

The type-name property of an element node indicates the relationship between its typed value and its string value. For example, if an element node has the type-name property `xs:decimal` and the string value "47.5", the typed value is the decimal value 47.5. If the type-name property of an element node is `xdt:untyped`, the element's typed value is equal to its string value and has the type `xdt:untypedAtomic`.

### Related concepts

"Node properties" on page 7

## Attribute nodes

An attribute node represents an XML attribute.

An attribute node can have zero or one parent. The element node that owns an attribute is considered to be its parent, even though an attribute node is not a child of its parent element.

An attribute node has the following node properties:

- node-name
- parent, possibly empty
- type-name
- string-value
- typed-value

Attribute nodes can be constructed in XQuery expressions by using direct or computed constructors.

The type-name property of an attribute node indicates the relationship between its typed value and its string value. For example, if an attribute node has the type-name property `xs:decimal` and the string value "47.5", its typed value is the decimal value 47.5.

### Related concepts

"Node properties" on page 7

### Related reference

"Computed attribute constructors" on page 84

"Direct element constructors" on page 76

## Text nodes

A text node encapsulates XML character content.

A text node can have zero or one parent. Text nodes that are children of a document or element node never appear as adjacent siblings. When a document or element node is constructed, any adjacent text node siblings are combined into a single text node. If the resulting text node is empty, it is discarded.

Text nodes have the following node properties:

- content, possibly empty
- parent, possibly empty

Text nodes can be constructed in XQuery expressions by computed constructors, or by the action of a direct element constructor.

### Related concepts

“Node properties” on page 7

### Related reference

“Text node constructors” on page 86

## Processing instruction nodes

A processing instruction node encapsulates an XML processing instruction.

A processing instruction node can have zero or one parent. The content of a processing instruction cannot include the string ?>. The target of a processing instruction must be an NCName. (The target is used to identify the application to which the instruction is directed.)

A processing instruction node has the following node properties:

- target
- content
- parent, possibly empty

Processing instruction nodes can be constructed in XQuery expressions by using direct or computed constructors.

### Related concepts

“Node properties” on page 7

### Related reference

“Processing instruction constructors” on page 86

## Comment nodes

A comment node encapsulates an XML comment.

A comment node can have zero or one parent. The content of a comment node cannot include the string "--" (two hyphens) or contain the hyphen character ( - ) as the last character.

A comment node has the following node properties:

- content
- parent, possibly empty

Comment nodes can be constructed in XQuery expressions by using direct or computed constructors.

### Related concepts

“Comments” on page 15

“Node properties” on page 7

### Related reference

“Comment constructors” on page 88

## Document order of nodes

All of the nodes in a hierarchy conform to an order, called *document order*, in which each node appears before its children. Document order corresponds to the order in which the nodes would appear if the node hierarchy were represented in serialized XML.

The nodes appear in the following order:

- The root node is the first node.
- Element nodes occur before their children.
- Attribute nodes immediately follow the element node with which they are associated. The relative order of attribute nodes is arbitrary, but this order does not change during the processing of a query.
- The relative order of siblings is determined by their order in the node hierarchy.
- Children and descendants of a node occur before siblings that follow the node.

### Related concepts

“Node hierarchies” on page 5

## Node identity

Each node has a unique identity. Two nodes are distinguishable even though their names and values might be the same. In contrast, atomic values do not have an identity.

Node identity is not the same as an ID-type attribute. An element in an XML document can be given an ID-type attribute by the document author. A node identity, however, is automatically assigned to every node by the system but is not directly visible to users.

Node identity is used to process the following types of expressions:

- Node comparisons. Node identity is used by the **is** operator to determine if two nodes have the same identity.
- Path expressions. Node identity is used by path expressions to eliminate duplicate nodes.
- Sequence expressions. Node identity is used by the **union**, **intersect**, or **except** operators to eliminate duplicate nodes.

### Related reference

“Node comparisons” on page 73

“Path expressions” on page 56

“Sequence expressions” on page 64

## Typed values and string values of nodes

Each node has both a *typed value* and a *string value*. These two node properties are used in the definitions of certain XQuery operations (such as atomization) and functions (such as `fn:data`, `fn:string`, and `fn:deep-equal`).

Table 1. String values and typed values of nodes

Node kind	String value	Typed value
Document	An instance of the <code>xs:string</code> data type that is the concatenated contents of all its descendant text nodes, in document order.	An instance of the <code>xdt:untypedAtomic</code> data type that is the concatenated contents of all its descendant text nodes, in document order.



Table 1. String values and typed values of nodes (continued)

Node kind	String value	Typed value
Element in a validated document	<ul style="list-style-type: none"> <li>• If validation assigned to the element a simple data type (such as xs:decimal) or a type that has simple content (such as a "temperature" type whose content is xs:decimal), the string value is the string that expresses the value of the element in the original XML document.</li> <li>• If validation assigned to the element a type that permits it to have mixed content (both text and child elements), the string value is an instance of the xs:string data type that is the concatenated contents of all its text node descendants, in document order.</li> <li>• If validation assigned to the element a type that permits no content (neither text nor child elements), the string value of the element is an empty string.</li> <li>• If validation assigned to the element a type that permits it to contain only child elements (no text), the string value of the element consists of the concatenated string values of all its text node descendants, in document order.</li> </ul>	<ul style="list-style-type: none"> <li>• If validation assigned to the element a simple data type (such as xs:decimal) or a type that has simple content (such as a "temperature" type whose content is xs:decimal), the typed value is the result of casting the string value to the simple type that is assigned by the validation process (for example, xs:decimal).</li> <li>• If validation assigned to the element a type that permits it to have mixed content (both text and child elements), the typed value is an instance of the xdt:untypedAtomic data type that is the concatenated contents of all its text node descendants, in document order.</li> <li>• If validation assigned to the element a type that permits no content (neither text nor child elements), the typed value is an empty sequence.</li> <li>• If validation assigned to the element a type that permits it to contain only child elements (no text), the element has no typed value, and an attempt to extract its typed value (for example, by the fn:data function) results in an error.</li> </ul>
Element in an unvalidated document	An instance of the xs:string data type that is the concatenated contents of all its text node descendants, in document order.	An instance of the xdt:untypedAtomic data type that is the concatenated contents of all its text node descendants, in document order.
Attribute in a validated document	An instance of the xs:string data type that represents the attribute value in the original XML document.	The result of casting the string value into the type that was assigned to the attribute during validation. For example, if an attribute is validated as having the type xs:decimal, its string value might be the string "74.8" and its typed value might be 74.8 as a decimal number.
Attribute in an unvalidated document	An instance of the xs:string data type that represents the attribute value in the original XML document.	An instance of the xdt:untypedAtomic data type that represents the attribute value in the original XML document.
Text	The content as an instance of the xs:string data type.	The content as an instance of the xdt:untypedAtomic data type.
Comment	The content as an instance of the xs:string data type.	The content as an instance of the xs:string data type.
Processing instruction	The content as an instance of the xs:string data type.	The content as an instance of the xs:string data type.

### Related concepts

"Node properties" on page 7

"Atomization" on page 50

"Document order of nodes" on page 10

"Node kinds" on page 7

---

## Serialization of the XDM

The result of an XQuery expression, which is an instance of the XDM, can be transformed into an XML representation through a process called *serialization*.

During serialization, the sequence of nodes and atomic values (the instance of the XDM) is converted into an XML representation. The result of serialization does not always represent a well-formed document. In fact, serialization can result in a single atomic value (for example, 17) or a sequence of elements that do not have a common parent.

XQuery does not provide a function to serialize the XDM. How the XDM is serialized into XML data depends on the environment in which the query is executing. For example, the CLP (command-line processor) returns a sequence of serialized items with each serialized item returned as a row in the result. For example, the query `XQUERY (1, 2, 3)`, when entered from the CLP, returns the following result:

```
1
2
3
```

Serialization can also be performed by the SQL/XML function `XMLSERIALIZE`.

#### **Related concepts**

“XQuery and XPath data model” on page 4

---

## **XML namespaces and QNames**

XML namespaces prevent naming collisions. An *XML namespace* is a collection of names that is identified by a namespace URI. Namespaces provide a way of qualifying names that are used for elements, attributes, data types, and functions in XQuery. A name that is qualified with a namespace prefix is a *qualified name (QName)*.

#### **Related reference**

“Namespace declaration” on page 44

“Namespace declaration attributes” on page 79

“In-scope namespaces of a constructed element” on page 82

## **Qualified names (QNames)**

A *QName* consists of an optional namespace prefix and a local name. The namespace prefix and the local name are separated by a colon. The namespace prefix, if present, is bound to a URI (Universal Resource Identifier) and provides a shortened form of the URI.

During query processing, XQuery expands the QName and resolves the URI that is bound to the namespace prefix. The expanded QName includes the namespace URI and a local name. Two QNames are equal if they have the same namespace URI and local name. This means that two QNames can match even if they have different prefixes provided that the prefixes are bound to the same namespace URI.

The following example includes the QNames:

- `ns1:name`
- `ns2:name`
- `name`

In this example, `ns1` is a prefix that is bound to the URI `http://posample.org`. The prefix `ns2` is bound to the URI `http://mycompany.com`. The default element namespace is another URI that is different from the URIs that are associated with `ns1` and `ns2`. The local name for all three elements is `name`.

```
<ns1:name>This text is in an element named "name" that is qualified
by the prefix "ns1".</ns1:name>
```

```
<ns2:name>This text is in an element named "name" that is qualified
by the prefix "ns2".</ns2:name>
```

```
<name>This text is in an element named "name" that is in the default
element namespace.</name>
```

The elements in this example share the same local name, name, but naming conflicts do not occur because the elements exist in different namespaces. During expression processing, the name ns1:name is expanded into a name that includes the URI that is bound to ns1 and the local name, name. Likewise, the name ns2:name is expanded into a name that includes the URI that is bound to ns2 and the local name, name. The element name, which has an empty prefix, is bound to the default element namespace because no prefix is specified. An error is returned if a name uses a prefix that is not bound to a URI.

QNames (qualified names) conform to the syntax that is defined in the W3C recommendation *Namespaces in XML*.

**Related concepts**

“Statically known namespaces”

**Related reference**

“Namespace declaration” on page 44

“resolve-QName function” on page 142

## Statically known namespaces

Namespace prefixes are bound to URIs by namespace declarations. The set of these namespace bindings that control the interpretation of QNames in a query expression is called the statically known namespaces. Statically known namespaces are properties of a query expression and are independent of the data that is processed by the expression.

Some namespace prefixes are predeclared; others can be added through declarations in either the query prolog or an element constructor. DB2 includes the predeclared namespace prefixes that are described in the following table.

*Table 2. Predeclared namespaces in DB2 XQuery*

Prefix	URI	Description
xml	http://www.w3.org/XML/1998/namespace	XML reserved namespace
xs	http://www.w3.org/2001/XMLSchema	XML Schema namespace
xsi	http://www.w3.org/2001/XMLSchema-instance	XML Schema instance namespace
fn	http://www.w3.org/2005/xpath-functions	Default function namespace
xdtd	http://www.w3.org/2005/xpath-datatypes	XQuery type namespace
db2-fn	http://www.ibm.com/xmlns/prod/db2/functions	DB2 function namespace

In addition to the predeclared namespaces, a set of statically known namespaces can be provided in the following ways:

- Declared in the query prolog, using either a namespace declaration or a default namespace declaration. The following example namespace declaration associates the namespace prefix ns1 with the URI http://mycompany.com:

```
declare namespace ns1 = "http://mycompany.com";
```

The following example default element/type namespace declaration sets the URI for element names in the query that do not have prefixes:

```
declare default element namespace "http://posample.org";
```

- Declared by a namespace declaration attribute in an element constructor. The following example is an element constructor that contains a namespace declaration attribute that binds the prefix ns2 to the URI http://mycompany.com within the scope of the constructed element:

```
<ns2:price xmlns:ns2="http://mycompany.com">14.99</ns2:price>
```

- Provided by SQL/XML. SQL/XML can provide the following set of namespaces:

- SQL/XML predeclared namespaces.
- Namespaces that are declared within SQL/XML constructors and other SQL/XML expressions.

Namespaces that are provided by SQL/XML can be overridden by namespace declarations in the prolog, or subsequent namespace declaration attributes in element constructors. Namespaces that are declared in the prolog can be overridden by namespace declaration attributes in element constructors.

**Related concepts**

“Qualified names (QNames)” on page 12

**Related reference**

“Namespace declaration” on page 44

“Namespace declaration attributes” on page 79

“In-scope namespaces of a constructed element” on page 82

---

## Language conventions

XQuery language conventions are described in the following topics.

### Case sensitivity

XQuery is a case-sensitive language.

Keywords in XQuery use lowercase characters and are not reserved. Names in XQuery expressions can be the same as language keywords.

**Related concepts**

“Retrieving DB2 data with XQuery functions” on page 2

### Whitespace

Whitespace is allowed in most XQuery expressions to improve readability even if whitespace is not part of the syntax for the expression. Whitespace consists of space characters (X'20'), carriage returns (X'0D'), line feeds (X'0A'), and tabs (X'09').

In general, whitespace is not significant in a query, except in the following situations where whitespace is preserved:

- The whitespace is in a string literal.
- The whitespace clarifies an expression by preventing the parser from recognizing two adjacent tokens as one.
- The whitespace is in an element constructor. The boundary-space declaration in the prolog determines whether to preserve or strip whitespace in element constructors.

For example, the following expressions require whitespace for clarity:

- `name- name` results in an error. The parser recognizes `name-` as a single QName (qualified name) and returns an error when no operator is found.
- `name -name` does not result in an error. The parser recognizes the first `name` as a QName, the minus sign (`-`) as an operator, and then the second `name` as another QName.
- `name-name` does not result in an error. However, the expression is parsed as a single QName because a hyphen (`-`) is a valid character in a QName.
- The following expressions all result in errors:
  - `10 div3`
  - `10div3`

In these expressions, whitespace is required for the parser to recognize each token separately.

**Related reference**

“Boundary whitespace in direct element constructors” on page 80

“Boundary-space declaration” on page 40

## Comments

Comments are allowed in the prolog or query body. Comments do not affect query processing.

A comment is composed of a string that is delimited by the symbols (: and :). The following example is a comment in XQuery:

```
(: A comment. You can use comments to make your code easier to understand. :)
```

The following general rules apply to using comments in DB2 XQuery:

- Comments can be used wherever ignorable whitespace is allowed. *Ignorable whitespace* is whitespace that is not significant to the expression results.
- Comments are not allowed in constructor content.
- Comments can nest within each other, but each nested comment must have open and close delimiters, (: and :).

The following examples illustrate legal comments and comments that result in errors:

- (: is this a comment? ::) is a legal comment.
- (: is this a comment? ::) or an error? :) results in an error because there is an unbalanced nesting of the symbols (: and :).
- (: commenting out a (: comment :) may be confusing, but often helpful :) is a legal comment because a balanced nesting of comments is allowed.
- "this is just a string :)" is a legal expression.
- (: "this is just a string :)") :) results in an error. Likewise, "this is another string (::" is a legal expression, but (: "this is another string (::" :) results in an error. Literal content can result in an unbalanced nesting of comments.

### Related concepts

“Comment nodes” on page 9

### Related reference

“Comment constructors” on page 88

---

## Where to find more information about XQuery

See these resources for more information about the specifications on which DB2 XQuery is based.

- **XQuery 1.0**  
World Wide Web Consortium. *XQuery 1.0: An XML Query Language*. W3C Candidate Recommendation, 03 November 2005. See [www.w3.org/TR/2005/CR-xquery-20051103/](http://www.w3.org/TR/2005/CR-xquery-20051103/).
- **XQuery 1.0 and XPath 2.0 Functions and Operators**  
World Wide Web Consortium. *XQuery 1.0 and XPath 2.0 Functions and Operators*. W3C Candidate Recommendation, 03 November 2005. See [www.w3.org/TR/2005/CR-xpath-functions-20051103/](http://www.w3.org/TR/2005/CR-xpath-functions-20051103/).
- **XQuery 1.0 and XPath 2.0 Data Model**  
World Wide Web Consortium. *XQuery 1.0 and XPath 2.0 Data Model*. W3C Candidate Recommendation, 3 November 2005. See [www.w3.org/TR/2005/CR-xpath-datamodel-20051103/](http://www.w3.org/TR/2005/CR-xpath-datamodel-20051103/).
- **XML Query Use Cases**  
World Wide Web Consortium. *XML Query Use Cases*. W3C Working Draft, 15 September 2005. See [www.w3.org/TR/xquery-use-cases/](http://www.w3.org/TR/xquery-use-cases/).
- **XML Schema**

World Wide Web Consortium. *XML Schema, Parts 0, 1, and 2*. W3C Recommendation, 2 May 2001. See [www.w3.org/TR/2001/REC-xmlschema-0-20010502/](http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/), [www.w3.org/TR/2001/REC-xmlschema-1-20010502/](http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/), and [www.w3.org/TR/2001/REC-xmlschema-2-20010502/](http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/).

- **XML Names**

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

---

## Chapter 2. Type system

XQuery is a strongly-typed language in which the operands of various expressions, operators, and functions must conform to expected types. The type system for DB2 XQuery includes the built-in types of XML Schema and the predefined types of XQuery.

The built-in types of XML Schema are in the namespace `http://www.w3.org/2001/XMLSchema`, which has the predeclared namespace prefix `xs`. Some examples of built-in schema types include `xs:integer`, `xs:string`, and `xs:date`.

The predefined types of XQuery are in the namespace `http://www.w3.org/2005/xpath-datatypes`, which has the predeclared namespace prefix `xdt`. Some examples of predefined types of XQuery include `xdt:untypedAtomic`, `xdt:yearMonthDuration`, and `xdt:dayTimeDuration`.

Each data type has a lexical form, which is a string that can be cast into the given type or that can be used to represent a value of the given type after serialization.

### **Related reference**

“Limits for XQuery data types” on page 161

---

## The type hierarchy

The DB2 XQuery type hierarchy shows all of the types that can be used in XQuery expressions.

The hierarchy in Figure 3 on page 18 includes abstract base types and derived types. All atomic types derive from the data type `xdt:anyAtomicType`. Solid lines connect each derived data type to the base types from which it is derived.

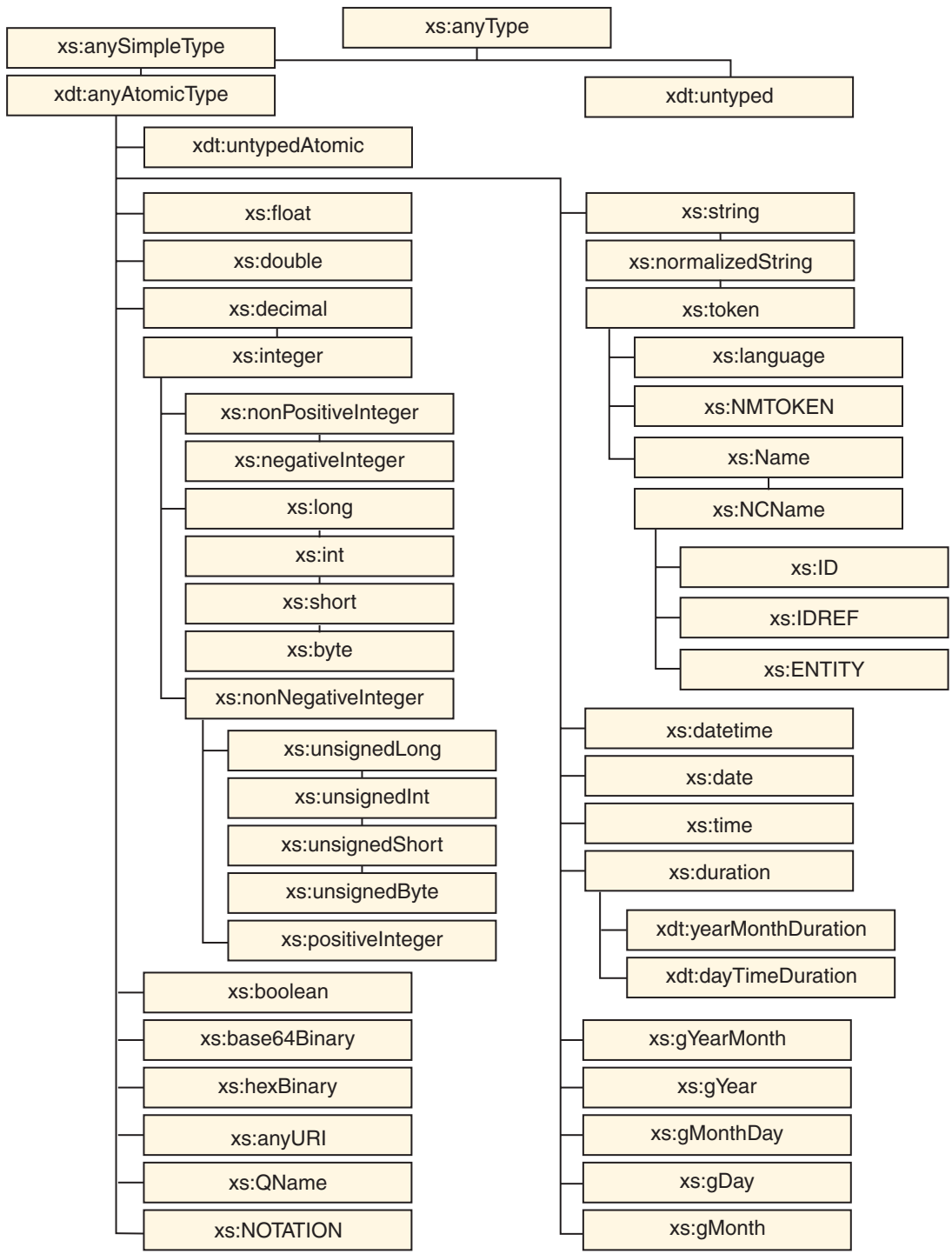


Figure 3. DB2 XQuery type hierarchy

## Types by category

DB2 XQuery has the following categories of types: generic, untyped, string, numeric, date, time, duration, and others.



## Generic data types

Table 3. Generic data types

Type	Description
“anyType data type” on page 26	The data type <code>xs:anyType</code> encompasses any sequence of zero or more nodes and zero or more atomic values.
“anySimpleType data type” on page 25	The data type <code>xs:anySimpleType</code> denotes a context where any simple type can be used. This data type serves as the base type for all simple types. An instance of a simple type may be any sequence of atomic values.
“anyAtomicType data type” on page 25	The data type <code>xdt:anyAtomicType</code> denotes a context where any atomic type can be used. This data type serves as the base type for all atomic types. An instance of an atomic type is a single nondecomposable value such as an integer, a string, or a date.

## Untyped data types

Table 4. Untyped data types

Type	Description
“untyped data type” on page 37	The data type <code>xdt:untyped</code> denotes a node that has not been validated by an XML schema.
“untypedAtomic data type” on page 37	The data type <code>xdt:untypedAtomic</code> denotes an atomic value that has not been validated by an XML schema.

## String data types

Table 5. String data types

Type	Description
“string data type” on page 36	The data type <code>xs:string</code> represents a character string.
“normalizedString data type” on page 35	The data type <code>xs:normalizedString</code> represents a whitespace-normalized string.
“token data type” on page 37	The data type <code>xs:token</code> represents a tokenized string.
“language data type” on page 34	The data type <code>xs:language</code> represents a natural language identifier as defined by <i>RFC 3066</i> .
“NMTOKEN data type” on page 35	The data type <code>xs:NMTOKEN</code> represents the NMTOKEN attribute type from <i>XML 1.0 (Third Edition)</i> .
“Name data type” on page 34	The data type <code>xs:Name</code> represents an XML Name.
“NCName data type” on page 34	The data type <code>xs:NCName</code> represents an XML noncolonized name.
“ID data type” on page 33	The data type <code>xs:ID</code> represents the ID attribute type from <i>XML 1.0 (Third Edition)</i> .
“IDREF data type” on page 33	The data type <code>xs:IDREF</code> represents the IDREF attribute type from <i>XML 1.0 (Third Edition)</i> .
“ENTITY data type” on page 31	The data type <code>xs:ENTITY</code> represents the ENTITY attribute type from <i>XML 1.0 (Third Edition)</i> .

## Numeric data types

Table 6. Numeric data types

Type	Description
“decimal data type” on page 29	The data type xs:decimal represents a subset of the real numbers that can be represented by decimal numerals.
“float data type” on page 31	The data type xs:float is patterned after the IEEE single-precision 32-bit floating point type.
“double data type” on page 29	The data type xs:double is patterned after the IEEE double-precision 64-bit floating point type.
“int data type” on page 34	The data type xs:int represents an integer that is less than or equal to 2 147 483 647 and greater than or equal to -2 147 483 648.
“nonPositiveInteger data type” on page 35	The data type xs:nonPositiveInteger represents an integer that is less than or equal to zero.
“negativeInteger data type” on page 34	The data type xs:negativeInteger represents an integer that is less than zero.
“nonNegativeInteger data type” on page 35	The data type xs:nonNegativeInteger represents an integer that is greater than or equal to zero.
“long data type” on page 34	The data type xs:long represents an integer that is less than or equal to 9 223 372 036 854 775 807 and greater than or equal to -9 223 372 036 854 775 808.
“integer data type” on page 34	The data type xs:integer represents a number that is less than or equal to 9 223 372 036 854 775 807 and greater than or equal to -9 223 372 036 854 775 808.
“short data type” on page 36	The data type xs:short represents an integer that is less than or equal to 32 767 and greater than or equal to -32 768.
“byte data type” on page 26	The data type xs:byte represents an integer that is less than or equal to 127 and greater than or equal to -128.
“unsignedLong data type” on page 37	The data type xs:unsignedLong represents an unsigned integer that is less than or equal to 9 223 372 036 854 775 807.
“unsignedInt data type” on page 37	The data type xs:unsignedInt represents an unsigned integer that is less than or equal to 4 294 967 295.
“unsignedShort data type” on page 37	The data type xs:unsignedShort represents an unsigned integer that is less than or equal to 65 535.
“unsignedByte data type” on page 37	The data type xs:unsignedByte represents an unsigned integer that is less than or equal to 255.
“positiveInteger data type” on page 35	The data type xs:positiveInteger represents a positive integer that is greater than or equal to 1.

## Date, time, and duration data types

Table 7. Date, time, and duration data types

Type	Description
“duration data type” on page 30	The data type xs:duration represents a duration of time that is expressed by the Gregorian year, month, day, hour, minute, and second components.

Table 7. Date, time, and duration data types (continued)

Type	Description
“yearMonthDuration data type” on page 38	The data type xdt:yearMonthDuration represents a duration of time that is expressed by the Gregorian year and month components.
“dayTimeDuration data type” on page 28	The data type xdt:dayTimeDuration represents a duration of time that is expressed by days, hours, minutes, and seconds components.
“dateTime data type” on page 27	The data type xs:dateTime represents an instant that has the following properties: year, month, day, hour, and minute properties that are expressed as integer values; a second property that is expressed as a decimal value; and an optional time zone indicator.
“date data type” on page 26	The data type xs:date represents an interval of exactly one day in duration that begins on the first moment of a given day. The data type xs:date consists of year, month, and day properties that are expressed as integer values and an optional timezone indicator.
“time data type” on page 36	The data type xs:time represents an instant of time that recurs every day.
“gYearMonth data type” on page 33	The data type xs:gYearMonth represents a specific Gregorian month in a specific Gregorian year. Gregorian calendar months are defined in <i>ISO 8601</i> .
“gYear data type” on page 32	The data type xs:gYear represents a Gregorian calendar year. Gregorian calendar years are defined in <i>ISO 8601</i> .
“gMonthDay data type” on page 32	The data type xs:gMonthDay represents a Gregorian date that recurs. Gregorian calendar dates are defined in <i>ISO 8601</i> .
“gDay data type” on page 31	The data type xs:gDay represents a Gregorian day that recurs. Gregorian calendar days are defined in <i>ISO 8601</i> .
“gMonth data type” on page 32	The data type xs:gMonth represents a Gregorian month that recurs every year. Gregorian calendar months are defined in <i>ISO 8601</i> .

## Other data types

Table 8. Other data types

Type	Description
“boolean data type” on page 26	The data type xs:boolean supports the mathematical concept of binary-valued logic: true or false.
“anyURI data type” on page 26	The data type xs:anyURI represents a Uniform Resource Identifier (URI).
“QName data type” on page 35	The data type xs:QName represents an XML qualified name (QName). A QName includes an optional namespace prefix, a URI that identifies the XML namespace, and a local part, which is an NCName.
“NOTATION data type” on page 35	The data type xs:NOTATION represents the NOTATION attribute type from <i>XML 1.0 (Third Edition)</i> .
“hexBinary data type” on page 33	The data type xs:hexBinary represents hex-encoded binary data.

Table 8. Other data types (continued)

Type	Description
“base64Binary data type” on page 26	The data type xs:base64Binary represents base64-encoded binary data.

**Related reference**

“Type casting” on page 23

## Constructor functions for built-in data types

*Constructor functions* convert an instance of one atomic type into an instance of a different atomic type. An implicitly-defined constructor function exists for each of the built-in atomic types that are defined in XML Schema. Constructor functions also exist for the data type xdt:untypedAtomic and the two derived data types xdt:yearMonthDuration and xdt:dayTimeDuration.

Constructor functions are not available for xs:NOTATION, xs:anyType, xs:anySimpleType, or xdt:anyAtomicType.

All constructor functions for built-in types share the following generic syntax:

▶▶—*type-name*(*value*)—▶▶

**Note:** The semantics of the constructor function *type-name*(*value*) are defined to be equivalent to the expression (*value* cast as *type-name*?).

*type-name*

The QName of the target data type.

*value*

The value to be constructed as an instance of the target data type. Atomization is applied to the value. If the result of atomization is an empty sequence, the empty sequence is returned. If the result of atomization is a sequence of more than one item, an error is raised. Otherwise, the resulting atomic value is cast to the target type. For information about which types can be cast to which other types, see “Type casting” on page 23.

For example, the following diagram represents the syntax of the constructor function for the XML Schema data type xs:unsignedInt:

▶▶—xs:unsignedInt(*value*)—▶▶

The value that can be passed to this constructor function is any atomic value that can be validly cast into the target data type. For example, the following invocations of this function return the same result, the xs:unsignedInt value 12:

```
xs:unsignedInt(12)
xs:unsignedInt("12")
```

In the first example, the numeric literal 12 is passed to the constructor function. Because the literal does not contain a decimal point, it is parsed as an xs:integer, and the xs:integer value is cast to the type xs:unsignedInt. In the second example, the string literal “12” is passed to the constructor function. The string literal is parsed as an xs:string, and the xs:string value is cast to the type xs:unsignedInt.

A constructor function can also be invoked with a node as its argument. In this case, DB2 XQuery atomizes the node to extract its typed value and then calls the constructor with that value. If the value that is passed to a constructor cannot be cast to the target data type, an error is returned.

The constructor function for `xs:QName` differs from the generic syntax for constructor functions in that the constructor function is constrained to take a string literal as its argument.

**Related concepts**

“Atomization” on page 50

**Related reference**

“Cast expressions” on page 102

“Type casting”

## Type casting

Type conversions are supported between `xdt:untypedAtomic`, `xs:integer`, the two derived types of `xs:duration` (`xdt:yearMonthDuration` and `xdt:dayTimeDuration`), and the nineteen primitive types that are defined in XML Schema. Type conversions are used in cast expressions and type constructors.

The type conversions that are supported are indicated in the following tables. Each table shows the primitive types that are the source of the type conversion on the left side and the primitive types that are the target of the type conversion on the top. The first table contains the targets from `xdt:untypedAtomic` to `xs:dateTime`, and the second table contains the targets from `xs:time` to `xs:NOTATION`.

The cells in the tables contain one of three characters:

- Y** Yes. Indicates that a conversion from values of the source type to the target type is supported.
- N** No. Indicates that a conversion from values of the source type to the target type is not supported.
- M** Maybe. Indicates that a conversion from values of the source type to the target type might succeed for some values and fail for other values.

Casting is not supported to or from `xs:anySimpleType` or to or from `xdt:anyAtomicType`.

If an unsupported casting is attempted, an error is raised.

*Table 9. Primitive type casting, part 1 (targets from `xdt:untypedAtomic` to `xs:dateTime`)*

Source data type	Target data type									
	uA	string	float	double	decimal	integer	dur	yMD	dTD	dT
uA	Y	Y	M	M	M	M	M	M	M	M
string	Y	Y	M	M	M	M	M	M	M	M
float	Y	Y	Y	Y	M	M	N	N	N	N
double	Y	Y	M	Y	M	M	N	N	N	N
decimal	Y	Y	Y	Y	Y	M	N	N	N	N
integer	Y	Y	Y	Y	Y	Y	N	N	N	N
dur	Y	Y	N	N	N	N	Y	Y	Y	N
yMD	Y	Y	N	N	N	N	Y	Y	N	N
dTD	Y	Y	N	N	N	N	Y	N	Y	N
dT	Y	Y	N	N	N	N	N	N	N	Y
time	Y	Y	N	N	N	N	N	N	N	N
date	Y	Y	N	N	N	N	N	N	N	Y
gYM	Y	Y	N	N	N	N	N	N	N	N
gYr	Y	Y	N	N	N	N	N	N	N	N
gMD	Y	Y	N	N	N	N	N	N	N	N

Table 9. Primitive type casting, part 1 (targets from xdt:untypedAtomic to xs:dateTime) (continued)

Source data type	Target data type									
	uA	string	float	double	decimal	integer	dur	yMD	dTD	dT
gDay	Y	Y	N	N	N	N	N	N	N	N
gMon	Y	Y	N	N	N	N	N	N	N	N
bool	Y	Y	Y	Y	Y	Y	N	N	N	N
b64	Y	Y	N	N	N	N	N	N	N	N
hxB	Y	Y	N	N	N	N	N	N	N	N
aURI	Y	Y	N	N	N	N	N	N	N	N
QN	Y	Y	N	N	N	N	N	N	N	N
NOT	Y	Y	N	N	N	N	N	N	N	N

Table 10. Primitive type casting, part 2 (targets from xs:time to xs:NOTATION)

Source data type	Target data type												
	time	date	gYM	gYr	gMD	gDay	gMon	bool	b64	hxB	aURI	QN	NOT
uA	M	M	M	M	M	M	M	M	M	M	M	N	N
string	M	M	M	M	M	M	M	M	M	M	M	M	M
float	N	N	N	N	N	N	N	Y	N	N	N	N	N
double	N	N	N	N	N	N	N	Y	N	N	N	N	N
decimal	N	N	N	N	N	N	N	Y	N	N	N	N	N
integer	N	N	N	N	N	N	N	Y	N	N	N	N	N
dur	N	N	N	N	N	N	N	N	N	N	N	N	N
yMD	N	N	N	N	N	N	N	N	N	N	N	N	N
dTD	N	N	N	N	N	N	N	N	N	N	N	N	N
dT	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N
time	Y	N	N	N	N	N	N	N	N	N	N	N	N
date	N	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N
gYM	N	N	Y	N	N	N	N	N	N	N	N	N	N
gYr	N	N	N	Y	N	N	N	N	N	N	N	N	N
gMD	N	N	N	N	Y	N	N	N	N	N	N	N	N
gDay	N	N	N	N	N	Y	N	N	N	N	N	N	N
gMon	N	N	N	N	N	N	Y	N	N	N	N	N	N
bool	N	N	N	N	N	N	N	Y	N	N	N	N	N
b64	N	N	N	N	N	N	N	N	Y	Y	N	N	N
hxB	N	N	N	N	N	N	N	N	Y	Y	N	N	N
aURI	N	N	N	N	N	N	N	N	N	N	Y	N	N
QN	N	N	N	N	N	N	N	N	N	N	N	N	N
NOT	N	N	N	N	N	N	N	N	N	N	N	N	M

The columns and rows are identified by short codes that identify the following types:

- uA = xdt:untypedAtomic

- string = xs:string
- float = xs:float
- double = xs:double
- decimal = xs:decimal
- integer = xs:integer
- dur = xs:duration
- yMD = xdt:yearMonthDuration
- dTD = xdt:dayTimeDuration
- dT = xs:dateTime
- time = xs:time
- date = xs:date
- gYM = xs:gYearMonth
- gYr = xs:gYear
- gMD = xs:gMonthDay
- gDay = xs:gDay
- gMon = xs:gMonth
- bool = xs:boolean
- b64 = xs:base64Binary
- hxB = xs:hexBinary
- aURI = xs:anyURI
- QN = xs:QName
- NOT = xs:NOTATION

#### **Related concepts**

“Constructor functions for built-in data types” on page 22

#### **Related reference**

“Cast expressions” on page 102

“Limits for XQuery data types” on page 161

---

## **anyAtomicType data type**

The data type `xdt:anyAtomicType` denotes a context where any atomic type can be used. This data type serves as the base type for all atomic types. An instance of an atomic type is a single nondecomposable value such as an integer, a string, or a date.

The data type `xdt:anyAtomicType` has an unconstrained lexical form.

Casting is not supported to or from the data type `xdt:anyAtomicType`.

---

## **anySimpleType data type**

The data type `xs:anySimpleType` denotes a context where any simple type can be used. This data type serves as the base type for all simple types. An instance of a simple type may be any sequence of atomic values.

The data type `xs:anySimpleType` has an unconstrained lexical form.

Casting is not supported to or from the data type `xs:anySimpleType`.

---

## anyType data type

The data type `xs:anyType` encompasses any sequence of zero or more nodes and zero or more atomic values.

---

## anyURI data type

The data type `xs:anyURI` represents a Uniform Resource Identifier (URI).

The lexical form of the data type `xs:anyURI` is a string that is a legal URI as defined by *RFC 2396* and amended by *RFC 2732*. Avoid using spaces in values of this type unless the spaces are encoded by `%20`.

---

## base64Binary data type

The data type `xs:base64Binary` represents base64-encoded binary data.

For base64-encoded binary data, the entire binary stream is encoded by using the base64 alphabet. The base64 alphabet is described in *RFC 2045*.

The lexical form of `xs:base64Binary` is limited to the 65 characters of the base64 alphabet that is defined in *RFC 2045*. Valid characters include a-z, A-Z, 0-9, the plus sign (+), the forward slash (/), the equal sign (=), and the characters defined in *XML 1.0 (Third Edition)* as white space. No other characters are allowed.

---

## boolean data type

The data type `xs:boolean` supports the mathematical concept of binary-valued logic: true or false.

The lexical form of the data type `xs:boolean` is constrained to the following values: true, false, 1, and 0.

---

## byte data type

The data type `xs:byte` represents an integer that is less than or equal to 127 and greater than or equal to -128.

The lexical form of `xs:byte` is an optional sign that is followed by a finite-length sequence of decimal digits. If the sign is omitted, a positive sign (+) is assumed. The following numbers are valid examples of this data type: -1, 0, 126, and +100.

---

## date data type

The data type `xs:date` represents an interval of exactly one day in duration that begins on the first moment of a given day. The data type `xs:date` consists of year, month, and day properties that are expressed as integer values and an optional timezone indicator.

Time-zoned values of type `xs:date` track the starting moment of the day, as determined by the timezone. The first moment of the day begins at 00:00:00, and the day continues until, but does not include, 24:00:00, which is the first moment of the following day. For example, the first moment of the date 2002-10-10+13:00 is the value 2002-10-10T00:00:00+13:00. This value is equivalent to 2002-10-09T11:00:00Z, which is also the first moment of 2002-10-09-11:00. Therefore, the values 2002-10-10+13:00 and 2002-10-09-11:00 represent the same interval.

The lexical form of `xs:date` is a finite-length sequence of characters of the following form: `yyyy-mm-ddzzzzzz`. Negative dates are not allowed. The following abbreviations are used to describe this form:



*yyyy*

A 4-digit numeral that represents the year. Valid values are from 0001 through 9999. A plus sign (+) is not allowed.

*mm*

A 2-digit numeral that represents the month.

*dd* A 2-digit numeral that represents the day.

*zzzzzz*

Optional. If present, represents the timezone. See “Timezone indicator” on page 28 for more information about the format for this property.

---

## dateTime data type

The data type `xs:dateTime` represents an instant that has the following properties: year, month, day, hour, and minute properties that are expressed as integer values; a second property that is expressed as a decimal value; and an optional time zone indicator.

Valid lexical representations of `xs:dateTime` might not have an explicit time zone. For representations that do not have an explicit time zone, an implicit time zone of UTC (Coordinated Universal Time, also called Greenwich Mean Time) is used. Each property expressed as a numeric value is constrained to the maximum value within the interval that is determined by the next-higher property. For example, the day value can never be 32 and cannot even be 29 for month 02 and year 2002 (February 2002).

The lexical form of `xs:dateTime` is a finite-length sequence of characters of the following form: `yyyy-mm-ddThh:mm:ss.ssssszzzzz`. Negative dates are not allowed. The following abbreviations describe this form:

*yyyy*

A 4-digit numeral that represents the year. Valid values are from 0001 through 9999. A plus sign (+) is not allowed.

- Separators between parts of the date portion

*mm*

A 2-digit numeral that represents the month.

*dd* A 2-digit numeral that represents the day.

**T** A separator to indicate that the time of day follows.

*hh* A 2-digit numeral that represents the hour. A value of 24 is allowed only when the minutes and seconds that are represented are zero. A query that includes the time of 24:00:00 is treated as 00:00:00 of the next day.

: A separator between parts of the time portion.

*mm*

A 2-digit numeral that represents the minute.

*ss* A 2-digit numeral that represents the whole seconds.

*.sssss*

Optional. If present, a 1-to-6 digit numeral that represents the fractional seconds.

*zzzzzz*

Optional. If present, represents the timezone. See “Timezone indicator” on page 28 for more information about the format for this property.

For example, the following form indicates noon on 10 October 2005, Eastern Standard Time in the United States:

2005-10-10T12:00:00-05:00

This time is expressed in UTC as 2002-10-10T17:00:00Z.

## Timezone indicator

The lexical form for the timezone indicator is a string that includes one of the following forms:

- A positive (+) or negative (-) sign that is followed by *hh:mm*, where the following abbreviations are used:

*hh* A 2-digit numeral (with leading zeros as required) that represents the hours. Currently, no legally prescribed time zones have durations greater than 24 hours. Therefore, a value of 24 for the hours property is allowed only when the value of the minutes property is zero.

*mm*

A 2-digit numeral that represents the minutes. The value of the minutes property must be zero when the hours property is equal to 14.

- + Indicates that the specified time instant is in a time zone that is ahead of the UTC time by *hh* hours and *mm* minutes.
  - Indicates that the specified time instant is in a time zone that is behind UTC time by *hh* hours and *mm* minutes.
- The literal Z, which represents the time in UTC (Z represents Zulu time, which is equivalent to UTC). Specifying Z for the time zone is equivalent to specifying +00:00 or -00:00.

---

## dayTimeDuration data type

The data type `xdtd:dayTimeDuration` represents a duration of time that is expressed by days, hours, minutes, and seconds components.

The range that can be represented by this data type is from -P8333333333333333Y3M11574074074DT1H46M39.999999S to P8333333333333333Y3M11574074074DT1H46M39.999999S (or -9999999999999999 months and -9999999999999999.999999 seconds to 9999999999999999 months and 9999999999999999.999999 seconds).

The lexical form of `xdtd:dayTimeDuration` is *PnDTnHnMnS*, which is a reduced form of the *ISO 8601* format. The following abbreviations describe this form:

**P** The duration designator.

**nD**

*n* is an unsigned integer that represents the number of days.

**T** The date and time separator.

**nH**

*n* is an unsigned integer that represents the number of hours.

**nM**

*n* is an unsigned integer that represents the number of minutes.

**nS**

*n* is an unsigned decimal that represents the number of seconds. If a decimal point appears, it must be followed by one to six digits that represent fractional seconds.

For example, the following form indicates a duration of 3 days, 10 hours, and 30 minutes:

P3DT10H30M

The following form indicates a duration of negative 120 days:

-P120D

An optional preceding minus sign (-) indicates a negative duration. If the sign is omitted, a positive duration is assumed.

Reduced precision and truncated representations of this format are allowed, but they must conform to the following requirements:

- If the number of days, hours, minutes, or seconds in any expression equals zero, the number and its corresponding designator can be omitted. However, at least one number and its designator must be present.
- The seconds part can have a decimal fraction.
- The designator T must be absent if and only if all of the time items are absent. The designator P must always be present.

For example, the following forms are allowed:

```
P13D
PT47H
P3DT2H
-PT35.89S
P4D251M
```

The form P-134D is not allowed, but the form -P1347D is allowed.

---

## decimal data type

The data type `xs:decimal` represents a subset of the real numbers that can be represented by decimal numerals.

The lexical form of `xs:decimal` is a finite-length sequence of decimal digits that are separated by a period as a decimal indicator. An optional leading sign is allowed. If the sign is omitted, a positive sign (+) is assumed. Leading and trailing zeroes are optional. If the fractional part is zero, the period and any following zeroes can be omitted. The following numbers are valid examples of this data type:

```
-1.23
12678967.543233
+100000.00
210
```

---

## double data type

The data type `xs:double` is patterned after the IEEE double-precision 64-bit floating point type.

The basic value space of `xs:double` consists of values that range from  $-1.7976931348623158e+308$  to  $-2.2250738585072014e-308$  and from  $+2.2250738585072014e-308$  to  $+1.7976931348623158e+308$ . The value space of `xs:double` also includes the following special values: positive infinity, negative infinity, positive zero, negative zero, and not-a-number (NaN).

The lexical form of `xs:double` is a mantissa followed, optionally, by the character E or e, followed by an exponent. The exponent must be an integer. The mantissa must be a decimal number. The representations for the exponent and the mantissa must follow the lexical rules for `xs:integer` and `xs:decimal`. If the E or e and the exponent that follows are omitted, an exponent value of 0 is assumed.

Lexical forms for zero can take a positive or negative sign. The following literals are valid examples of this data type: `-1E4`, `1267.43233E12`, `12.78e-2`, `12`, `-0`, and `0`.

The special values positive infinity, negative infinity, and not-a-number have the lexical forms `INF`, `-INF` and `NaN`, respectively. The lexical form for positive infinity cannot take a positive sign.

**Tip:** There is no literal for the special values INF, -INF and NaN. Construct the values INF, -INF, and NaN from strings by using the xs:double type constructor. For example: xs:doubl1e("INF").

**Related reference**

“number function” on page 137

---

## duration data type

The data type xs:duration represents a duration of time that is expressed by the Gregorian year, month, day, hour, minute, and second components.

The range that can be represented by this data type is from -P8333333333333333Y3M11574074074DT1H46M39.999999S to P8333333333333333Y3M11574074074DT1H46M39.999999S (or -9999999999999999 months and -9999999999999999.999999 seconds to 9999999999999999 months and 9999999999999999.999999 seconds).

The lexical form of xs:duration is the *ISO 8601* extended format *PnYnMnDTnHnMnS*. The following abbreviations describe the extended format:

**P** The duration designator.

*nY*

*n* is an unsigned integer that represents the number of years.

*nM*

*n* is an unsigned integer that represents the number of months.

*nD*

*n* is an unsigned integer that represents the number of days.

**T** The date and time separator.

*nH*

*n* is an unsigned integer that represents the number of hours.

*nM*

*n* is an unsigned integer that represents the number of minutes.

*nS*

*n* is an unsigned decimal that represents the number of seconds. If a decimal point appears, it must be followed by one to six digits that represent fractional seconds.

For example, the following form indicates a duration of 1 year, 2 months, 3 days, 10 hours, and 30 minutes:

P1Y2M3DT10H30M

The following form indicates a duration of negative 120 days:

-P120D

An optional preceding minus sign (-) indicates a negative duration. If the sign is omitted, a positive duration is assumed.

Reduced precision and truncated representations of this format are allowed, but they must conform to the following requirements:

- If the number of years, months, days, hours, minutes, or seconds in any expression equals zero, the number and its corresponding designator can be omitted. However, at least one number and its designator must be present.
- The seconds part can have a decimal fraction.

- The designator T must be absent if and only if all of the time items are absent.
- The designator P must always be present.

For example, the following forms are allowed:

```
P1347Y
P1347M
P1Y2MT2H
P0Y1347M
P0Y1347M0D
```

The form P1Y2MT is not allowed because no time items are present. The form P-1347M is not allowed, but the form -P1347M is allowed.

---

## ENTITY data type

The data type `xs:ENTITY` represents the ENTITY attribute type from *XML 1.0 (Third Edition)*.

The lexical form of `xs:ENTITY` is an XML name that does not contain a colon (NCName).

---

## float data type

The data type `xs:float` is patterned after the IEEE single-precision 32-bit floating point type.

The basic value space of `xs:float` consists of values that range from  $-3.4028234663852886e+38$  to  $-1.1754943508222875e-38$  and from  $+1.1754943508222875e-38$  to  $+3.4028234663852886e+38$ . The value space of `xs:float` also includes the following special values: positive infinity, negative infinity, positive zero, negative zero, and not-a-number (NaN).

The lexical form of `xs:float` is a mantissa followed, optionally, by the character E or e, followed by an exponent. The exponent must be an integer. The mantissa must be a decimal number. The representations for the exponent and the mantissa must follow the lexical rules for `xs:integer` and `xs:decimal`. If the E or e and the exponent that follows are omitted, an exponent value of 0 is assumed.

Lexical forms for zero can take a positive or negative sign. The following literals are valid examples of this data type: `-1E4`, `1267.43233E12`, `12.78e-2`, `12`, `-0`, and `0`.

The special values positive infinity, negative infinity, and not-a-number have the lexical forms `INF`, `-INF` and `NaN`, respectively. The lexical form for positive infinity cannot take a positive sign.

**Tip:** There is no literal for the special values `INF`, `-INF` and `NaN`. Construct the values `INF`, `-INF`, and `NaN` from strings by using the `xs:float` type constructor. For example: `xs:float("INF")`.

---

## gDay data type

The data type `xs:gDay` represents a Gregorian day that recurs. Gregorian calendar days are defined in *ISO 8601*.

This data type represents a specific day of the month. For example, this data type might be used to indicate that payday is the 15th of each month.

The lexical form of `xs:gDay` is `---ddzzzzz`, which is a truncated representation of `xs:date` that does not include the month or year properties. No preceding sign is allowed. No other formats are allowed. The following abbreviations describe this form:

*dd* A 2-digit numeral that represents the day.

*ZZZZZZ*

Optional. If present, represents the timezone. See “Timezone indicator” on page 28 for more information about the format for this property.

For example, the following form indicates the sixteenth of the month, which is a day that recurs every month:

---16

---

## **gMonth data type**

The data type `xs:gMonth` represents a Gregorian month that recurs every year. Gregorian calendar months are defined in *ISO 8601*.

This data type represents a specific month of the year. For example, this data type might be used to indicate that Christmas is celebrated in the month of December.

The lexical form of `xs:gMonth` is `--mmzzzzzz`, which is a truncated representation of `xs:date` that does not include the year or day properties. No preceding sign is allowed. No other formats are allowed. The following abbreviations describe this form:

*mm*

A 2-digit numeral that represents the month.

*ZZZZZZ*

Optional. If present, represents the timezone. See “Timezone indicator” on page 28 for more information about the format for this property.

For example, the following form indicates December, a specific month that recurs every year:

--12

---

## **gMonthDay data type**

The data type `xs:gMonthDay` represents a Gregorian date that recurs. Gregorian calendar dates are defined in *ISO 8601*.

This data type represents a specific day of the year. For example, this data type might be used to indicate a birthday that occurs on the 16th of April every year.

The lexical form of `xs:gMonthDay` is `--mm-ddzzzzzz`, which is a truncated representation of `xs:date` that does not include the year property. No preceding sign is allowed. No other formats are allowed. The following abbreviations are used to describe this form:

*mm*

A 2-digit numeral that represents the month.

*dd* A 2-digit numeral that represents the day.

*ZZZZZZ*

Optional. If present, represents the timezone. See “Timezone indicator” on page 28 for more information about the format for this property.

For example, the following form indicates April 16, a specific day that recurs every year:

--04-16

---

## **gYear data type**

The data type `xs:gYear` represents a Gregorian calendar year. Gregorian calendar years are defined in *ISO 8601*.

The lexical form of `xs:gYear` is `yyyyzzzzzz`. This form is a truncated representation of `xs:dateTime` that does not include the month, day, or time of day properties. Negative dates are not allowed. The following abbreviations describe this form:

*yyyy*

A 4-digit numeral that represents the year. Valid values are from 0001 through 9999. A plus sign (+) is not allowed.

*zzzzzz*

Optional. If present, represents the timezone. See “Timezone indicator” on page 28 for more information about the format for this property.

For example, the following form represents the Gregorian year 2005: 2005.

---

## gYearMonth data type

The data type `xs:gYearMonth` represents a specific Gregorian month in a specific Gregorian year. Gregorian calendar months are defined in *ISO 8601*.

The lexical form of `xs:gYearMonth` is `yyyy-mmzzzzzz`. This form is a truncated representation of `xs:dateTime` that does not include the time of day properties. Negative dates are not allowed. The following abbreviations describe this form:

*yyyy*

A 4-digit numeral that represents the year. Valid values are from 0001 through 9999. A plus sign (+) is not allowed.

*mm*

A 2-digit numeral that represents the month.

*zzzzzz*

Optional. If present, represents the timezone. See “Timezone indicator” on page 28 for more information about the format for this property.

For example, the following form, which does not include an optional timezone indicator, indicates the month of October in 2005:

2005-10

---

## hexBinary data type

The data type `xs:hexBinary` represents hex-encoded binary data.

The lexical form of `xs:hexBinary` is a sequence of characters in which each binary octet is represented by two hexadecimal digits. For example, the following form is a hex encoding for the 16-bit integer 4023, which has a binary representation of 111110110111: 0FB7.

---

## ID data type

The data type `xs:ID` represents the ID attribute type from *XML 1.0 (Third Edition)*.

The lexical form of `xs:ID` is an XML name that does not contain a colon (NCName).

---

## IDREF data type

The data type `xs:IDREF` represents the IDREF attribute type from *XML 1.0 (Third Edition)*.

The lexical form of `xs:IDREF` is an XML name that does not contain a colon (NCName).

---

## int data type

The data type `xs:int` represents an integer that is less than or equal to 2 147 483 647 and greater than or equal to -2 147 483 648.

The lexical form of `xs:int` is an optional sign that is followed by a finite-length sequence of decimal digits. If the sign is omitted, a positive sign (+) is assumed. The following numbers are valid examples of this data type: -1, 0, 126789675, and +100000.

---

## integer data type

The data type `xs:integer` represents a number that is less than or equal to 9 223 372 036 854 775 807 and greater than or equal to -9 223 372 036 854 775 808.

The lexical form of `xs:integer` is a finite-length sequence of decimal digits with an optional leading sign. If the sign is omitted, a positive sign (+) is assumed. The following numbers are valid examples of this data type: -1, 0, 12678967543233, and +100000.

---

## language data type

The data type `xs:language` represents a natural language identifier as defined by *RFC 3066*.

The lexical form of `xs:language` consists of strings of tags connected by hyphens. Each tag contains no more than eight characters. The first tag can contain only alphabetic characters, and subsequent tags can contain alphabetic and numeric characters. For example, the value `en-US` represents the English language as used in the United States. The string conforms to the pattern `[a-zA-Z]{1,8}(-[a-zA-Z0-9]{1,8})*`.

---

## long data type

The data type `xs:long` represents an integer that is less than or equal to 9 223 372 036 854 775 807 and greater than or equal to -9 223 372 036 854 775 808.

The lexical form of `xs:long` is an optional sign that is followed by a finite-length sequence of decimal digits. If the sign is omitted, a positive sign (+) is assumed. The following numbers are valid examples of this data type: -1, 0, 12678967543233, and +100000.

---

## Name data type

The data type `xs>Name` represents an XML Name.

The lexical form of `xs>Name` is a string that matches the Name production of *XML 1.0 (Third Edition)*.

---

## NCName data type

The data type `xs:NCName` represents an XML noncolonized name.

The lexical form of `xs:NCName` is an XML name that does not contain a colon.

---

## negativeInteger data type

The data type `xs:negativeInteger` represents an integer that is less than zero.

The lexical form of `xs:negativeInteger` is negative sign (-) that is followed by a finite-length sequence of decimal digits. The range that can be represented by this data type is from -9223372036854775808 to -1. The following numbers are valid examples of this data type: -1, -12678967543233, and -100000.



---

## NMTOKEN data type

The data type `xs:NMTOKEN` represents the NMTOKEN attribute type from *XML 1.0 (Third Edition)*.

The lexical form of `xs:NMTOKEN` is a string that matches the `Nmtoken` production of *XML 1.0 (Third Edition)*.

---

## nonNegativeInteger data type

The data type `xs:nonNegativeInteger` represents an integer that is greater than or equal to zero.

The lexical form of `xs:nonNegativeInteger` is an optional sign that is followed by a finite-length sequence of decimal digits. If the sign is omitted, a positive sign (+) is assumed. For lexical forms that denote zero, the sign may be positive (+) or negative (-). In all other lexical forms, the sign, if present, must be positive (+). The range that can be represented by this data type is from 0 to +9223372036854775807. The following numbers are valid examples of this data type: 1, 0, 12678967543233, and +100000.

---

## nonPositiveInteger data type

The data type `xs:nonPositiveInteger` represents an integer that is less than or equal to zero.

The lexical form of `xs:nonPositiveInteger` is an optional preceding sign that is followed by a finite-length sequence of decimal digits. For lexical forms that denote zero, the sign may be negative (-) or may be omitted; in all other lexical forms, the negative sign (-) must be present. The range that can be represented by this data type is from -9223372036854775808 to 0. The following numbers are valid examples of this data type: -1, 0, -12678967543233, and -100000.

---

## normalizedString data type

The data type `xs:normalizedString` represents a whitespace-normalized string.

The lexical form of `xs:normalizedString` is a string that does not contain the carriage return (X'0D'), line feed (X'0A'), or tab (X'09') characters.

---

## NOTATION data type

The data type `xs:NOTATION` represents the NOTATION attribute type from *XML 1.0 (Third Edition)*.

The lexical form of the data type `xs:NOTATION` is the lexical form of the type `xs:QName`.

### Related reference

“QName data type”

---

## positiveInteger data type

The data type `xs:positiveInteger` represents a positive integer that is greater than or equal to 1.

The lexical form of `xs:positiveInteger` is an optional positive sign (+) that is followed by a finite-length sequence of decimal digits. The range that can be represented by this data type is from +1 to +9223372036854775807. The following numbers are valid examples of this data type: 1, 12678967543233, and +100000.

---

## QName data type

The data type `xs:QName` represents an XML qualified name (QName). A QName includes an optional namespace prefix, a URI that identifies the XML namespace, and a local part, which is an NCName.

The lexical form of the data type `xs:QName` is a string of the following format: *prefix:localName*. The following abbreviations are used to describe this form:

*prefix*

Optional. A namespace prefix. The namespace prefix must be bound to a URI reference by a namespace declaration. The prefix functions only as a placeholder for a namespace name. If no prefix is specified, the URI for the default element/type namespace is used.

*localName*

An NCName that is the local part of the qualified name. An NCName is an XML name without a colon.

For example, the following string is a valid lexical form of a QName that includes a prefix:

`ns1:emp`

---

## short data type

The data type `xs:short` represents an integer that is less than or equal to 32 767 and greater than or equal to -32 768.

The lexical form of `xs:short` is an optional sign that is followed by a finite-length sequence of decimal digits. If the sign is omitted, a positive sign (+) is assumed. The following numbers are valid examples of this data type: -1, 0, 12678, and +10000.

---

## string data type

The data type `xs:string` represents a character string.

The lexical form of `xs:string` is a sequence of characters that can include any character that is in the range of legal characters for XML.

---

## time data type

The data type `xs:time` represents an instant of time that recurs every day.

The lexical form of `xs:time` is *hh:mm:ss.sssssszzzzzz*. This form is a truncated representation of `xs:dateTime` that does not include the year, day, or month properties. The following abbreviations describe this form:

*hh* A 2-digit numeral that represents the hour. A value of 24 is allowed only when the minutes and seconds that are represented are zero. A query that includes the time of 24:00:00 is treated as 00:00:00 of the next day.

*:* A separator between parts of the time portion.

*mm*

A 2-digit numeral that represents the minute.

*ss* A 2-digit numeral that represents the whole seconds.

*.sssss*

Optional. If present, a 1-to-6 digit numeral that represents the fractional seconds.

*zzzzzz*

Optional. If present, represents the timezone. See “Timezone indicator” on page 28 for more information about the format for this property.

For example, the following form, which includes an optional timezone indicator, represents 1:20 pm Eastern Standard Time, which is 5 hours earlier than Coordinated Universal Time (UTC):

`13:20:00-05:00`

---

## token data type

The data type `xs:token` represents a tokenized string.

The lexical form of `xs:token` is a string that does not contain any of the following characters:

- carriage return (X'0D')
- line feed (X'0A')
- tab (X'09')
- leading or trailing spaces (X'20')
- internal sequences of two or more spaces

---

## unsignedByte data type

The data type `xs:unsignedByte` represents an unsigned integer that is less than or equal to 255.

The lexical form of `xs:unsignedByte` is a finite-length sequence of decimal digits. The following numbers are valid examples of this data type: 0, 126, and 100.

---

## unsignedInt data type

The data type `xs:unsignedInt` represents an unsigned integer that is less than or equal to 4 294 967 295.

The lexical form of `xs:unsignedInt` is a finite-length sequence of decimal digits. The following numbers are valid examples of this data type: 0, 1267896754, and 100000.

---

## unsignedLong data type

The data type `xs:unsignedLong` represents an unsigned integer that is less than or equal to 9 223 372 036 854 775 807.

The lexical form of `xs:unsignedLong` is a finite-length sequence of decimal digits. The following numbers are valid examples of this data type: 0, 12678967543233, and 100000.

---

## unsignedShort data type

The data type `xs:unsignedShort` represents an unsigned integer that is less than or equal to 65 535.

The lexical form of `xs:unsignedShort` is a finite-length sequence of decimal digits. The following numbers are valid examples of this data type: 0, 12678, and 10000.

---

## untyped data type

The data type `xdt:untyped` denotes a node that has not been validated by an XML schema.

If an element node is annotated as `xdt:untyped`, then all of its descendant element nodes are also annotated as `xdt:untyped`.

---

## untypedAtomic data type

The data type `xdt:untypedAtomic` denotes an atomic value that has not been validated by an XML schema.

The data type `xdt:untypedAtomic` has an unconstrained lexical form.

---

## yearMonthDuration data type

The data type `xd:yearMonthDuration` represents a duration of time that is expressed by the Gregorian year and month components.

The range that can be represented by this data type is from `-P8333333333333333Y3M` to `P8333333333333333Y3M` (or `-9999999999999999` to `9999999999999999` months).

The lexical form of `xd:yearMonthDuration` is `PnYnM`, which is a reduced form of the *ISO 8601* format. The following abbreviations describe this form:

### **nY**

*n* is an unsigned integer that represents the number of years.

### **nM**

*n* is an unsigned integer that represents the number of months.

An optional preceding minus sign (-) indicates a negative duration. If the sign is omitted, a positive duration is assumed.

For example, the following form indicates a duration of 1 year and 2 months:

`P1Y2M`

The following form indicates a duration of negative 13 months:

`-P13M`

Reduced precision and truncated representations of this format are allowed, but they must conform to the following requirements:

- The designator `P` must always be present.
- If the number of years or months in any expression equals zero, the number and its corresponding designator can be omitted. However, at least one number and its designator (`Y` or `M`) must be present.

For example, the following forms are allowed:

`P1347Y`

`P1347M`

The form `P-1347M` is not allowed, but the form `-P1347M` is allowed. The forms `P24YM` and `PY43M` are not allowed because `Y` must have at least one preceding digit and `M` must have one preceding digit.

## Chapter 3. Prolog

The *prolog* is series of declarations that define the processing environment for a query. Each declaration in the prolog is followed by a semicolon (;). The prolog is an optional part of the query; a valid query can consist of a query body with no prolog.

The prolog includes an optional version declaration, namespace declarations, and *setters*, which are optional declarations that set the values of properties that affect query processing.

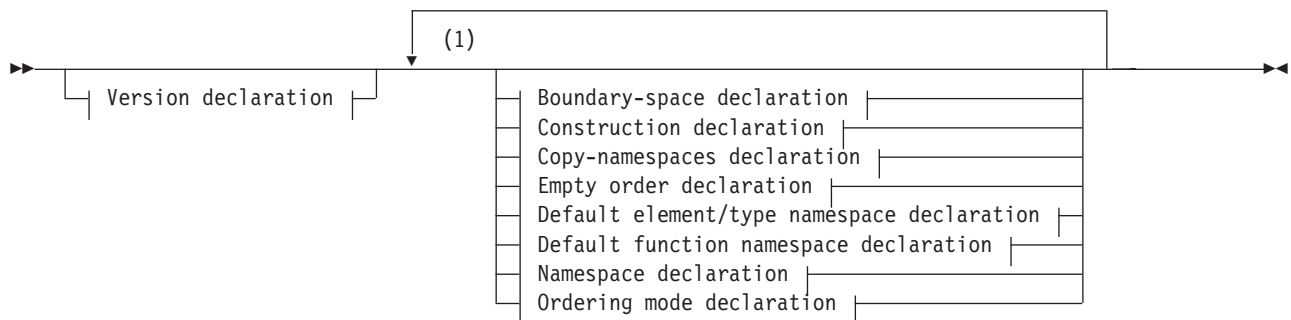
DB2 supports the boundary-space declaration that can be used to change how the query is processed. The prolog also consists of namespace declarations and default namespace declarations.

DB2 also supports the following setters. However, they do not change the processing environment because DB2 supports only one option in each case:

- Construction declaration
- Copy-namespaces declaration
- Empty order declaration
- Ordering mode declaration

The version declaration, if present, must be first in the prolog. Setters and other declarations can appear in any order in the prolog after the version declaration.

### Syntax



### Notes:

- 1 Each declaration can be specified only once, except for the namespace declaration.

#### Related concepts

“Introduction to XQuery” on page 1

---

## Version declaration

A version declaration appears at the beginning of a query to identify the version of the XQuery syntax and semantics that are needed to process the query. The version declaration can include an encoding declaration, but the encoding declaration is ignored by DB2.

If present, the version declaration must be at the beginning of the prolog. The only version that is supported by DB2 is “1.0”.

### Syntax

►► xquery version "1.0" encoding *StringLiteral* ;

## 1.0

Specifies that version 1.0 of the XQuery syntax and semantics is needed to process the query.

### *StringLiteral*

Specifies a string literal that represents the encoding name. Specifying an encoding declaration has no effect on the query because the value of *StringLiteral* is ignored. DB2 always assumes the encoding is UTF-8.

## Example

The following version declaration indicates that the query must be processed by an implementation that supports XQuery Version 1.0:

```
xquery version "1.0";
```

---

## Boundary-space declaration

A boundary-space declaration in the query prolog sets the boundary-space policy for the query. The *boundary-space policy* controls how boundary whitespace is processed by element constructors. *Boundary whitespace* includes all whitespace characters that occur by themselves in the boundaries between tags or enclosed expressions in element constructors.

The boundary-space policy can specify that boundary whitespace is either preserved or stripped (removed) when elements are constructed. If no boundary-space declaration is specified, the default behavior is to strip boundary whitespace when elements are constructed.

The prolog can contain only one boundary-space declaration for a query.

## Syntax

►► declare boundary-space strip  
preserve ;

### **strip**

Specifies that boundary whitespace is removed when elements are constructed.

### **preserve**

Specifies that boundary whitespace is preserved when elements are constructed.

## Example

The following boundary-space declaration specifies that boundary whitespace is preserved when elements are constructed:

```
declare boundary-space preserve;
```

### **Related concepts**

“Whitespace” on page 14

### **Related reference**

“Boundary whitespace in direct element constructors” on page 80

---

## Construction declaration

A construction declaration in the query prolog sets the construction mode for the query. The *construction mode* controls how type annotations are assigned to element and attribute nodes that are copied to form the content of a newly constructed node.

In DB2 XQuery, the construction mode for constructed element nodes is always **preserve**. When the construction mode is **preserve**, the copied attributes and descendants of the constructed element retain their original types.

A construction declaration that specifies a value other than **preserve** results in an error. The prolog can contain only one construction declaration for a query.

## Syntax

```
►► declare construction preserve ; ◀◀
```

### **preserve**

Specifies that the copied attributes and descendants of the constructed element retain their original types.

## Example

The following construction declaration is valid, but does not change the default behavior for element construction:

```
declare construction preserve;
```

### **Related concepts**

Chapter 2, “Type system,” on page 17

---

## Copy-namespaces declaration

The copy-namespaces mode controls the namespace bindings that are assigned when an existing element node is copied by an element constructor.

In DB2, the copy-namespaces mode is always **preserve** and **inherit**. The setting **preserve** specifies that all in-scope-namespaces of the original element are retained in the new copy. The default namespace is treated like any other namespace binding: the copied node preserves its default namespace or absence of a default namespace. The setting **inherit** specifies that the copied node inherits in-scope namespaces from the constructed node. In case of a conflict, the namespace bindings that were preserved from the original node take precedence.

A copy-namespaces declaration that specifies values other than **preserve** and **inherit** results in an error. The prolog can contain only one copy-namespaces declaration for a query.

## Syntax

```
►► declare copy-namespaces preserve , inherit ; ◀◀
```

### **preserve**

Specifies that all in-scope namespaces of the original element are retained in the new copy.

### **inherit**

Specifies that the copied node inherits in-scope namespaces from the constructed node.

## Example

The following copy-namespace declaration is valid, but does not change the default behavior for element construction:

```
declare copy-namespaces preserve, inherit;
```

### **Related reference**

“In-scope namespaces of a constructed element” on page 82

---

## Default element/type namespace declaration

The default element/type namespace declaration in the query prolog specifies the namespace to use for the unprefixes QNames (qualified names) of element and type names.

The query prolog can contain one default element/type namespace declaration only. This declaration is in scope throughout the query in which it is declared, unless the declaration is overridden by a namespace declaration attribute in a direct element constructor. If no default element/type namespace is declared, then unprefixes element and type names are not in any namespace.

The default element/type namespace does not apply to unqualified attribute names. Unprefixes attribute names and variable names are in no namespace.

### Syntax

► declare default element namespace *URILiteral*; ◀

#### element

Specifies that the declaration is a default element/type namespace declaration.

#### *URILiteral*

Specifies a string literal that represents the URI for the namespace. The string literal must be a valid URI or a zero-length string. If the string literal in a default element/type namespace declaration is a zero-length string, then unprefixes element and type names are not in any namespace.

### Example

The following declaration specifies that the default namespace for element and type names is the namespace that is associated with the URI `http://posample.org`:

```
declare default element namespace "http://posample.org";
<name>Snow boots</name>
```

When the query in the example executes, the newly created node (an element node called name) is in the namespace that is associated with the namespace URI `http://posample.org`.

#### Related concepts

“XML namespaces and QNames” on page 12

#### Related reference

“Namespace declaration” on page 44

“Namespace declaration attributes” on page 79

“Direct element constructors” on page 76

---

## Default function namespace declaration

The default function namespace declaration in the query prolog specifies a namespace URI that is used for unprefixes function names in function calls.

The query prolog can contain one default function namespace declaration only. If no default function namespace is declared, the default function namespace is the namespace of XPath and XQuery functions, `http://www.w3.org/2005/xpath-functions`. If you declare a default function namespace, you can invoke any function in the default function namespace without specifying a prefix.

DB2 returns an error if the local name for an unprefixes function call does not match a function in the default function namespace.



## Syntax

►►—declare—default—function—namespace—*URILiteral*—;—◄◄

### function

Specifies that the declaration is a default function namespace declaration

#### *URILiteral*

Specifies a string literal that represents the URI for the namespace. The string literal must be a valid URI or a zero-length string. If the string literal in a default function namespace declaration is a zero-length string, all function calls must use prefixed function names because every function is in some namespace.

## Example

The following declaration specifies that the default function namespace is associated with the URI `http://www.ibm.com/xmlns/prod/db2/functions`:

```
declare default function namespace "http://www.ibm.com/xmlns/prod/db2/functions";
```

Within the query body for this example, you could refer to any function in the default function namespace without including a prefix in the function name. This default function namespace includes the function `xmlcolumn`, so you can type `xmlcolumn('T1.MYDOC')` instead of typing `db2-fn:xmlcolumn('T1.MYDOC')`. However, because the default function namespace in this example is no longer associated with the namespace for XQuery functions, you would need to specify a prefix when you call XQuery built-in functions. For example, you must type `fn:current-date()` instead of typing `current-date()`.

#### Related concepts

Chapter 5, “Built-in functions,” on page 105

---

## Empty order declaration

An empty order declaration in the query prolog controls whether an empty sequence or a NaN value is interpreted as the greatest value or as the least value when an **order by** clause in a FLWOR expression is processed.

In DB2 XQuery, an empty sequence is always interpreted as the greatest value during processing of an **order by** clause in a FLWOR expression. A NaN value is interpreted as greater than all other values except an empty sequence. This setting cannot be overridden. An empty order declaration that specifies a value other than **empty greatest** results in an error. The query prolog can contain only one empty order declaration for a query.

## Syntax

►►—declare—default—order—empty—greatest—;—◄◄

### greatest

Specifies that an empty sequence is always interpreted as the greatest value during processing of an **order by** clause in a FLWOR expression. A NaN value is interpreted as greater than all other values except an empty sequence.

## Example

The following empty order declaration is valid:

```
declare default order empty greatest;
```

#### Related reference

“**order by** clauses” on page 95

---

## Ordering mode declaration

An *ordering mode declaration* in the query prolog sets the ordering mode for the query. The *ordering mode* defines the ordering of nodes in the query result.

Because DB2 does not support ordered mode as defined in *XQuery 1.0: An XML Query Language*, the ordering mode declaration, if present, must specify *unordered*. For the rules that govern the order of query results in DB2, see “Order of results in XQuery expressions” on page 48.

The query prolog can contain only one ordering mode declaration. An ordering mode declaration that specifies a value other than *unordered* results in an error.

### Syntax

►►—declare—ordering—unordered—;——————▶▶

#### *unordered*

Specifies that the rules for ordered mode in *XQuery 1.0: An XML Query Language* are not in effect. For the rules that govern the order of query results in DB2, see “Order of results in XQuery expressions” on page 48.

### Example

The following declaration is valid, but it does not change the default behavior of ordering because DB2 supports only *unordered* mode:

```
declare ordering unordered;
```

#### **Related reference**

“Order of results in XQuery expressions” on page 48

---

## Namespace declaration

A namespace declaration in the query prolog declares a namespace prefix and associates the prefix with a namespace URI. An association between a prefix and a namespace URI is called a *namespace binding*. A namespace that is bound in a namespace declaration is added to the statically known namespaces. The *statically known namespaces* consist of all of the namespace bindings that can be used to resolve namespace prefixes during the processing of a query.

The namespace declaration is in scope throughout the query in which it is declared, unless the declaration is overridden by a namespace declaration attribute in a direct element constructor. Multiple declarations of the same namespace prefix in the query prolog result in an error.

### Syntax

►►—declare—namespace—*prefix*—=—*URILiteral*—;——————▶▶

#### *prefix*

Specifies a namespace prefix that is bound to the URI that is specified by *URILiteral*. The namespace prefix is used in qualified names (QNames) to identify the namespace for an element, attribute, data type, or function.

The prefixes *xmlns* and *xml* are reserved and cannot be specified as prefixes in namespace declarations.

#### *URILiteral*

Specifies the URI to which the prefix is bound. *URILiteral* must be a non-zero-length literal string that contains a valid URI.

## Example

The following query includes a namespace declaration that declares the namespace prefix `ns1` and associates it with the namespace URI `http://posample.org`:

```
declare namespace ns1 = "http://posample.org";
<ns1:name>Thermal gloves</ns1:name>
```

When the query in the example executes, the newly created node (an element node called `name`) is in the namespace that is associated with the namespace URI `http://posample.org`.

## Predeclared namespace prefixes

XQuery has several predeclared namespace prefixes that are present in the statically known namespaces before each query is processed. You can use any of the predeclared prefixes without an explicit declaration. The predeclared namespace prefixes for DB2 XQuery include the prefix and URI pairs that are shown in the following table:

*Table 11. Predeclared namespaces in DB2 XQuery*

Prefix	URI	Description
xml	<a href="http://www.w3.org/XML/1998/namespace">http://www.w3.org/XML/1998/namespace</a>	XML reserved namespace
xs	<a href="http://www.w3.org/2001/XMLSchema">http://www.w3.org/2001/XMLSchema</a>	XML Schema namespace
xsi	<a href="http://www.w3.org/2001/XMLSchema-instance">http://www.w3.org/2001/XMLSchema-instance</a>	XML Schema instance namespace
fn	<a href="http://www.w3.org/2005/xpath-functions">http://www.w3.org/2005/xpath-functions</a>	Default function namespace
xdt	<a href="http://www.w3.org/2005/xpath-datatypes">http://www.w3.org/2005/xpath-datatypes</a>	XQuery type namespace
db2-fn	<a href="http://www.ibm.com/xmlns/prod/db2/functions">http://www.ibm.com/xmlns/prod/db2/functions</a>	DB2 function namespace

You can override predeclared namespace prefixes by specifying a namespace declaration in a query prolog. However, you cannot override the URI that is associated with the prefix `xml`.

### Related concepts

“XML namespaces and QNames” on page 12

### Related reference

“Namespace declaration attributes” on page 79

“Default element/type namespace declaration” on page 42



---

## Chapter 4. Expressions

Expressions are the basic building blocks of a query. Expressions can be used alone or in combination with other expressions to form complex queries. DB2 XQuery supports several kinds of expressions for working with XML data.

---

### Concepts for expression processing

The following topics describe certain operations that are often included in the processing of expressions. This includes extracting atomic values from nodes, using type promotion and subtype substitution to obtain values of an expected type, and computing the Boolean value of a sequence.

### Dynamic context and focus

The dynamic context of an expression is the information that is available at the time that the expression is evaluated. The focus, which consists of the context item, context position, and context size, is an important part of the dynamic context.

The focus changes as DB2 processes each item in a sequence. The focus consists of the following information:

#### Context item

The atomic value or node that is currently being processed. The context item can be retrieved by the context item expression, which consists of a single dot ( . ).

#### Context position

The position of the context item in the sequence that is currently being processed. The context item can be retrieved by the `fn:position()` function.

#### Context size

The number of items in the sequence that is currently being processed. The context size can be retrieved by the `fn:last()` function.

#### Related concepts

“Sequences and items” on page 4

#### Related reference

“Context item expressions” on page 55

“last function” on page 126

“position function” on page 138

### Precedence

The XQuery grammar defines a built-in precedence among operators and expressions. If an expression that has a lower precedence is used as an operand of an expression that has a higher precedence, the expression that has a lower precedence must be enclosed in parentheses.

The following table lists XQuery operators and expressions in order of their precedence from lowest to highest. The associativity column indicates the order in which operators or expressions of equal precedence are applied.

Table 12. Precedence in DB2 XQuery

Operator or expression	Associativity
, (comma)	left-to-right
:= (assignment)	right-to-left
FLWOR, some, every, if	left-to-right

Table 12. Precedence in DB2 XQuery (continued)

Operator or expression	Associativity
or	left-to-right
and	left-to-right
eq, ne, lt, le, gt, ge, =, !=, <, <=, >, >=, is, <<, >>	left-to-right
to	left-to-right
+, -	left-to-right
*, div, idiv, mod	left-to-right
union,	left-to-right
intersect, except	left-to-right
cast	left-to-right
-(unary), +(unary)	right-to-left
?	left-to-right
/, //	left-to-right
[ ], ( ), { }	left-to-right

#### Related reference

“Parenthesized expression” on page 55

“Arithmetic expressions” on page 67

## Order of results in XQuery expressions

In DB2, some kinds of expressions return sequences in a deterministic order while other kinds of expressions do not.

The following kinds of expressions return sequences in a deterministic order:

- FLWOR expressions that contain an explicit **order by** clause return results in the order specified. For example, the following expression returns a sequence of product elements in ascending order by price:

```
for $p in /product
order by $p/price
return $p
```

- Expressions that combine sequences with the **union**, **intersect**, or **except** operator return results in document order.
- Path expressions that satisfy the following conditions return results in document order:
  - The path expression contains only forward-axis steps.
  - The path expression has its origin in a single node, such as might result from a function call or a variable reference.
  - No step in the path expression contains more than a single predicate.
  - The path expression does not contain a fn:position function call or a fn:last function call.

The following example is a path expression that returns results in document order, assuming that the variable \$bib is bound to a single element.

```
$bib/book[title eq "War and Peace"]/chapter
```

- Range expressions, which are expressions that contain the **to** operator, return sequences of integers in ascending order. For example: 15 to 25.
- Expressions that contain comma operators, if all the operands are sequences with deterministic order, return results in the order of their operands. For example the following expression returns the sequence (5, 10, 15, 16, 17, 18, 19, 20, 25):  
(5, 10, 15 to 20, 25)

- Other expressions that contain operand expressions that return results in deterministic order return results in a deterministic order. For example, assuming the variable \$pub is bound to a single element, the following conditional expression returns ordered results because the path expressions in the then and else clauses return ordered results:

```
if ($pub/type eq "journal")
then $pub/editor
else $pub/author
```

If an expression that is not listed in the previous list returns more than one item, the order of items in the sequence is nondeterministic.

Table 13. Summary of ordering of results in XQuery expressions

Expression kind	Conditions for a deterministic ordering	Ordering of results	Example
FLWOR	Explicit <b>order by</b> clause	Determined by the <b>order by</b> clause	The following expression returns a sequence of product elements in ascending order by price:  for \$p in /product order by \$p/price return \$p
Expressions with <b>union</b> , <b>intersect</b> , or <b>except</b> operators	None	Document order	\$managers union \$students
Path expressions	<ul style="list-style-type: none"> <li>• The path expression contains only forward-axis steps.</li> <li>• The path expression has its origin in a single node, such as might result from a function call or a variable reference.</li> <li>• No step in the path expression contains more than a single predicate.</li> <li>• The path expression does not contain a fn:position function call or a fn:last function call.</li> </ul>	Document order	The following example is a path expression that returns results in document order, assuming that the variable \$bib is bound to a single element.  \$bib/book [title eq "War and Peace"] /chapter
Range expressions, which are expressions that contain the <b>to</b> operator	None	Sequence of integers in ascending order	15 to 25
Other expressions	Operand expressions all return results that are in a deterministic order	Determined by the ordering of the results of the nested expressions	Assuming the variable \$pub is bound to a single element, the following conditional expression returns ordered results because the path expressions in the then and else clauses return ordered results:  if (\$pub/type eq "journal") then \$pub/editor else \$pub/author

**Note:** If a positional predicate is applied to a sequence that does not have a deterministic order, the result is nondeterministic, which means that any item in the sequence can be selected.

#### **Related reference**

“Ordering mode declaration” on page 44

“**order by** clauses” on page 95

“Conditional expressions” on page 100

“Path expressions” on page 56

## **Atomization**

*Atomization* is the process of converting a sequence of items into a sequence of atomic values. Atomization is used by expressions whenever a sequence of atomic values is required.

Each item in a sequence is converted to an atomic value by applying the following rules:

- If the item is an atomic value, then the atomic value is returned.
- If the item is a node, then its typed value is returned. The *typed value* of a node is a sequence of zero or more atomic values that can be extracted from the node. If the node has no typed value, then an error is returned.

Implicit atomization of a sequence produces the same result as invoking the `fn:data` function explicitly on a sequence.

For example, the following sequence contains a combination of nodes and atomic values:

```
("Some text", <anElement xsi:type="string">More text</anElement>,
<anotherElement xsi:type="decimal">1.23</anotherElement>, 1001)
```

Applying atomization to this sequence results in the following sequence of atomic values:

```
("Some text", "More text", 1.23, 1001)
```

The following XQuery expressions use atomization to convert items into atomic values:

- Arithmetic expressions
- Comparison expressions
- Function calls with arguments whose expected types are atomic
- Cast expressions
- Constructor expressions for various kinds of nodes
- **order by** clauses in FLWOR expressions
- Type constructor functions

#### **Related concepts**

“Sequences and items” on page 4

“Atomic values” on page 4

“Typed values and string values of nodes” on page 10

#### **Related reference**

“data function” on page 117

## **Subtype substitution**

*Subtype substitution* is the use of a value whose dynamic type is derived from an expected type.

Subtype substitution does not change the actual type of a value. For example, if an `xs:integer` value is used where an `xs:decimal` value is expected, the value retains its type as `xs:integer`.

In the following example, the `fn:compare` function compares an `xs:string` value to an `xs:NCName` value:



```
fn:compare("product", xs:NCName("product"))
```

The returned value is 0, which means that the arguments compare as equal. Although the `fn:compare` function expects arguments of type `xs:string`, the function can be invoked with a value of type `xs:NCNAME` because this type is derived from `xs:string`.

Subtype substitution is used whenever an expression is passed a value that is derived from an expected type.

#### Related concepts

“The type hierarchy” on page 17

Chapter 2, “Type system,” on page 17

## Type promotion

*Type promotion* is a process that converts an atomic value from its original type to the type that is expected by an expression. XQuery uses type promotion during the evaluation of function calls, **order by** clauses, and operators that accept numeric or string operands.

XQuery permits the following type promotions:

#### Numeric type promotion:

A value of type `xs:float` (or any type that is derived by restriction from `xs:float`) can be promoted to the type `xs:double`. The result is the `xs:double` value that is the same as the original value.

A value of type `xs:decimal` (or any type that is derived by restriction from `xs:decimal`) can be promoted to either of the types `xs:float` or `xs:double`. The result of this promotion is created by casting the original value to the required type. This kind of promotion might cause loss of precision.

In the following example, a sequence that contains the `xs:double` value `13.54e-2` and the `xs:decimal` value `100` is passed to the `fn:sum` function, which returns a value of type `xs:double`:

```
fn:sum(xs:double(13.54e-2), xs:decimal(100))
```

#### URI type promotion:

A value of type `xs:anyURI` (or any type that is derived by restriction from `xs:anyURI`) can be promoted to the type `xs:string`. The result of this promotion is created by casting the original value to the type `xs:string`.

In the following example, the URI value is promoted to the expected type `xs:string`, and the function returns 18:

```
fn:string-length(xs:anyURI("http://example.com"))
```

Note that type promotion and subtype substitution differ in the following ways:

- For type promotion, the atomic value is actually converted from its original type to the type that is expected by an expression.
- For subtype substitution, an expression that expects a specific type can be invoked with a value that is derived from that type. However, the value retains its original type.

#### Related concepts

Chapter 2, “Type system,” on page 17

## Effective Boolean value

The *effective boolean value (EBV)* of a sequence is computed implicitly during the processing of expressions that require Boolean values. The EBV of a value is determined by applying the `fn:boolean` function to a value.

The following table describes the EBVs that are returned for specific types of values.

Table 14. EBVs returned for specific types of values in XQuery

Description of value	EBV returned
An empty sequence	false
A sequence whose first item is a node	true
A single value of type xs:boolean (or derived from xs:boolean)	false - if the xs:boolean value is false true - if the xs:boolean value is true
A single value of type xs:string or xdt:untypedAtomic (or derived from one of these types)	false - if the length of the value is zero true - if the length of the value is greater than zero
A single value of any numeric type (or derived from a numeric type)	false - if the value is NaN or is numerically equal to zero true - if the value is not numerically equal to zero
All other values	error

**Note:** The effective boolean value of a sequence that contains at least one node and at least one atomic value is nondeterministic in a query where the order is unpredictable.

The effective boolean value of a sequence is computed implicitly when the following types of expressions are processed:

- Logical expressions (**and**, **or**)
- The fn:not function
- The **where** clause of a FLWOR expression
- Certain types of predicates, such as a[b]
- Conditional expressions (**if**)
- Quantified expressions (**some**, **every**)

**Related reference**

- “boolean function” on page 111
- “Logical expressions” on page 73

---

## Primary expressions

Primary expressions are the basic primitives of the language. They include literals, variable references, parenthesized expressions, context item expressions, constructors, and function calls.

### Literals

A *literal* is a direct syntactic representation of an atomic value. DB2 XQuery supports two kinds of literals: numeric literals and string literals.

A *numeric literal* is an atomic value of type xs:integer, xs:decimal, or xs:double:

- A numeric literal that contains no decimal point (.) and no e or E character is an atomic value of type xs:integer. For example, 12 is a numeric literal.
- A numeric literal that contains a decimal point (.), but no e or E character is an atomic value of type xs:decimal. For example, 12.5 is a numeric literal.
- A numeric literal that contains an e or E character is an atomic value of type xs:double. For example, 125E2 is a numeric literal.

Values of numeric literals are interpreted according to the rules of XML Schema.

A *string literal* is an atomic value of type `xs:string` that is enclosed in delimiting single quotation marks (') or double quotation marks ("). String literals can include predefined entity references and character references. For example, the following strings are valid string literals:

```
"12.5"
"He said, ""Let it be."""
'She said: "Why should I?"'
"Ben & Jerry's"
"&#8364;65.50" (: denotes the string €65.50 :)
```

**Tip:** To include a single quotation mark within a string literal that is delimited by single quotation marks, specify two adjacent single quotation marks. Similarly, to include a double quotation mark within a string literal that is delimited by double quotation marks, specify two adjacent double quotation marks.

Within a string literal, line endings are normalized according to the rules for *XML 1.0 (Third Edition)*. Any two-character sequence that contains a carriage return (X'0D') followed by a line feed (X'0A') is translated into a single line feed (X'0A'). Any carriage return (X'0D') that is not followed by a line feed (X'0A') is translated into a single line feed (X'0A').

If the value that you want to instantiate has no literal representation, you can use a constructor function or built-in function to return the value. The following functions and constructors return values that have no literal representation:

- The built-in functions `fn:true()` and `fn:false()` return the boolean values `true` and `false`, respectively. These values can also be returned by the constructor functions `xs:boolean("false")` and `xs:boolean("true")`.
- The constructor function `xs:date("2005-04-16")` returns an item whose type is `xs:date` and whose value represents the date April 16, 2005.
- The constructor function `xdt:dayTimeDuration("PT4H")` returns an item whose type is `xdt:dayTimeDuration` and whose value represents a duration of four hours.
- The constructor function `xs:float("NaN")` returns the special floating-point value, "Not a Number."
- The constructor function `xs:double("INF")` returns the special double-precision value, "positive infinity."

#### Related concepts

"Atomic values" on page 4

"Constructor functions for built-in data types" on page 22

Chapter 5, "Built-in functions," on page 105

#### Related reference

"Size limits" on page 162

## Predefined entity references

A *predefined entity reference* is a short sequence of characters that represents a character that has some syntactic significance in DB2 XQuery. A predefined entity reference begins with an ampersand (&) and ends with a semicolon (;). When a string literal is processed, each predefined entity reference is replaced by the character that it represents.

The following table lists the predefined entity references that DB2 XQuery recognizes.

Table 15. Predefined entity references in DB2 XQuery

Entity reference	Character represented
&lt;	<
&gt;	>
&amp;	&
&quot;	"
&apos;	'

### Related reference

“Character references”

## Character references

A *character reference* is an XML-style reference to a Unicode character that is identified by its decimal or hexadecimal code point.

A character reference begins with either `&#x` or `&#`, and it ends with a semicolon (`;`). If the character reference begins with `&#x`, the digits and letters before the terminating semicolon (`;`) provide a hexadecimal representation of the character’s code point in the *ISO/IEC 10646* standard. If the character reference begins with `&#`, the digits before the terminating semicolon (`;`) provide a decimal representation of the character’s code point.

### Example

The character reference `&#8364;` or `&#x20AC;` represents the Euro symbol (€).

### Related reference

“Predefined entity references” on page 53

## Variable references

A variable reference is an NCName that is preceded by a dollar sign (`$`). When a query is evaluated, each variable reference resolves to the value that is bound to the variable. Every variable reference must match a name in the in-scope variables at the point of reference.

Variables are added to the in-scope variables in the following ways:

- A variable can be added to the in-scope variables by the host language environment, SQL/XML, through the XMLQUERY function, the XMLTABLE function, or the XMLEXISTS predicate. A variable that is added by SQL/XML is in scope for the entire query unless the variable is overridden by another binding of the same variable in an XQuery expression.
- A variable can be bound to a value by an XQuery expression. The kinds of expressions that can bind variables are FLWOR expressions and quantified expressions. Function calls also bind values to the formal parameters of functions before executing the function body. A variable that is bound by an XQuery expression is in scope throughout the expression in which it is bound.

A variable name cannot be declared more than once in a FLWOR expression. For example, DB2 does not support the following expression:

```
for $i in (1, 2)
for $i in ("a", "b")
return $i
```

If a variable reference matches two or more variable bindings that are in scope, then the reference refers to the inner binding (the binding whose scope is smaller).

**Tip:** To make your code easier to read, use unique names for variables within a query.

### Example

In the following example, a FLWOR expression binds the variable `$seq` to the sequence (10, 20, 30):

```
let $seq := (10, 20, 30)
return $seq[2];
```

The returned value is 20.

### Related reference

“FLWOR expressions” on page 89

“Quantified expressions” on page 101

## Parenthesized expression

Parentheses can be used to enforce a particular order of evaluation in expressions that contain multiple operators.

For example, the expression  $(2 + 4) * 5$  evaluates to thirty, because the parenthesized expression  $(2 + 4)$  is evaluated first, and its result is multiplied by five. Without parentheses, the expression  $2 + 4 * 5$  evaluates to twenty-two, because the multiplication operator has higher precedence than the addition operator.

Empty parentheses denote an empty sequence.

### Related reference

“Precedence” on page 47

“Arithmetic expressions” on page 67

## Context item expressions

A context item expression consists of a single period character (.). A context item expression evaluates to the item that is currently being processed, which is known as the *context item*. The context item can be either a node or an atomic value. Context items are defined only in path expressions and predicate expressions.

### Example

The following example contains a context item expression that invokes the modulus operator on every item in the sequence that is returned by the range expression 1 to 100:

```
(1 to 100)[. mod 5 eq 0]
```

The result of this example is the sequence of integers between 1 and 100 that are evenly divisible by 5.

### Related reference

“Dynamic context and focus” on page 47

“Predicates” on page 63

“Path expressions” on page 56

## Function calls

A function call consists of a QName that is followed by a parenthesized list of zero or more expressions, which are called arguments. DB2 XQuery supports calls to built-in XQuery functions and DB2-specific functions.

Built-in XQuery functions are in the namespace `http://www.w3.org/2005/xpath-functions`, which is bound to the prefix `fn`. DB2-specific functions are in the namespace `http://www.ibm.com/xmlns/prod/db2/functions`, which is bound to the prefix `db2-fn`. If the QName in the function call has no namespace prefix, the function must be in the default function namespace. The default function namespace is the namespace of built-in XQuery functions (bound to the prefix `fn`) unless the namespace is overridden by a default function declaration in the query prolog.

**Important:** Because the arguments of a function call are separated by commas, you must use parentheses to enclose argument expressions that contain top-level comma operators.

The following steps explain the process that DB2 XQuery uses to evaluate functions:

1. DB2 XQuery evaluates each expression that is passed as an argument in the function call and returns a value for each expression.

2. The value that is returned for each argument is converted to the data type that is expected for that argument. When the argument expects an atomic value or a sequence of atomic values, DB2 XQuery uses the following rules to convert the value of the argument to its expected type:
  - a. Atomization is applied to the given value. This results in a sequence of atomic values.
  - b. Each item in the atomic sequence that is of type `xdt:untypedAtomic` is cast to the expected atomic type. For built-in functions that expect numeric arguments, arguments of type `xdt:untypedAtomic` are cast to `xs:double`.
  - c. Numeric type promotion is applied to any numeric item in the atomic sequence that can be promoted to the expected atomic type. Numeric items include items of type `xs:integer` (or derived from `xs:integer`), `xs:decimal`, `xs:float`, or `xs:double`.
  - d. If the expected type is `xs:string`, each item in the atomic sequence that is of type `xs:anyURI`, or derived from `xs:anyURI`, is promoted to `xs:string`.
3. The function is evaluated using the converted values of its arguments. The result of the function call is either an instance of the function's declared return type or an error.

## Examples

**Function call with a string argument:** The following function call takes an argument of type `xs:string` and returns a value of type `xs:string` in which all characters are in uppercase:

```
fn:upper-case($ns1_customerinfo/ns1:addr/@country)
```

In this example, the argument that is passed to the `fn:upper-case` function is a path expression. When the function is invoked, the path expression is evaluated and the resulting node sequence is atomized. Each atomic value in the sequence is cast to the expected type, `xs:string`. The function is evaluated and returns a sequence of atomic values of type `xs:string`.

**Function call with a sequence argument:** The following function takes a sequence, (1, 2, 3), as the single argument.

```
fn:max((1, 2, 3))
```

Because the function `fn:max` expects a single argument that is a sequence of atomic values, nested parentheses are required. The returned value is 3.

### Related concepts

“XML namespaces and QNames” on page 12

Chapter 5, “Built-in functions,” on page 105

“Type promotion” on page 51

“Atomization” on page 50

---

## Path expressions

*Path expressions* identify nodes within an XML tree. Path expressions in DB2 XQuery are based on the syntax of XPath 2.0.

A path expression consists of one or more steps that are separated by slash (/) or double-slash (//) characters. A path expression can begin with a step or with a slash or double-slash character. Each step before the final step generates a sequence of nodes that are used as context nodes for the step that follows.

The first step specifies the starting point of the path, often by using a function call or variable reference that returns a node or sequence of nodes. An initial "/" indicates that the path begins at the root node of the tree that contains the context node. An initial "/" indicates that the path begins with an initial node sequence that consists of the root node of the tree that contains the context node, plus all of the descendants of the root node.

Each step is executed repeatedly, once for each context node that is generated by the previous step. The results of these repeated executions are then combined to form the sequence of context nodes for the step that follows. Duplicate nodes are eliminated from this combined sequence, based on node identity.

The value of the path expression is the combined sequence of items that results from the final step in the path. This value can be either a sequence of nodes or a sequence of atomic values. Because each step in a path provides context nodes for the step that follows, the final step in a path is the only step that can return a sequence of atomic values. A path expression that returns a mixture of nodes and atomic values results in an error.

The node sequence that results from a path expression is not guaranteed to be in a specific order. To understand when a path expression returns ordered results, see the topic that describes the order of results in XQuery expressions.

**Related concepts**

“Node hierarchies” on page 5

**Related reference**

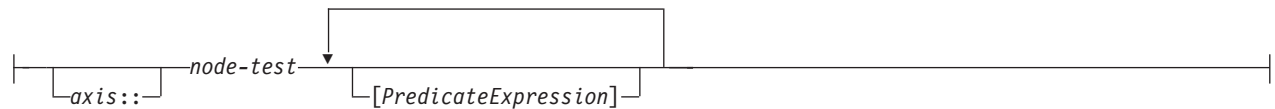
“Order of results in XQuery expressions” on page 48

## Syntax of path expressions

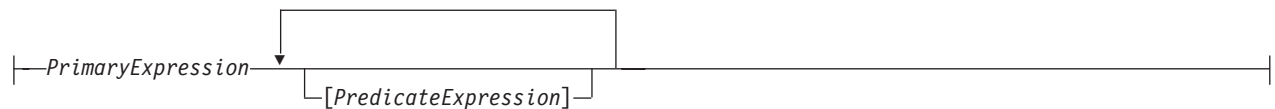
Each step of a path expression is either an axis step or a filter expression. An *axis step* returns a sequence of nodes that are reachable from the context node via a specified axis. A *filter expression* consists of a primary expression that is followed by zero or more predicates.



**axis step:**



**filter expression:**



/ An initial slash character (/) indicates that the path begins at the root node, which must be a document node, of the tree that contains the context node. Slash characters within a path expression separate steps.

// An initial double slash character (//) indicates that the path begins with an initial node sequence that consists of the root node, which must be a document node, of the tree that contains the context node, plus all of the descendants of the root node. To understand the meaning of a double slash character between steps, see the topic about abbreviated syntax.

*axis*

A direction of movement through an XML document or fragment. The list of supported axes includes child, descendant, attribute, self, descendant-or-self, and parent. Some of these axes can be represented by using an abbreviated syntax.

### *node-test*

A condition that must be true for each node that is selected by an axis step. This test can be either a name test that selects nodes based on the name of the node or a kind test that selects nodes based on the kind of node.

### *PrimaryExpression*

A primary expression.

### *PredicateExpression*

An expression that determines whether items of the sequence are retained or discarded.

## Examples

The following example shows an axis step that includes two predicates. This step selects all the employee children of the context node that have both a secretary child element and an assistant child element:

```
child::employee[secretary][assistant]
```

The following example uses a filter expression as a step in a path expression. The expression returns every chapter or appendix that contains more than one footnote within a given book:

```
$book/(chapter | appendix)[fn:count(footnote) > 1]
```

#### **Related reference**

“Axis steps”

“Abbreviated syntax for path expressions” on page 61

“Primary expressions” on page 52

“Predicates” on page 63

“Filter expressions” on page 65

## Axis steps

An *axis step* consists of three parts: an optional *axis*, which specifies a direction of movement; a *node test*, which specifies the criteria that is used to select nodes; and zero or more *predicates*, which filter the sequence that is returned by the step. The result of an axis step is always a sequence of zero or more nodes.

An axis step can be either a *forward step*, which starts at the context node and moves through the XML tree in document order, or a *reverse step*, which starts at the context node and moves through the XML tree in reverse document order. If the context item is not a node, then the expression results in an error.

The unabbreviated syntax for an axis step consists of an axis name and node test that are separated by a double colon, followed by zero or more predicates. The syntax of an axis expression can be abbreviated by omitting the axis and using shorthand notations.

In the following example, `child` is the name of the axis and `para` is the name of the element nodes to be selected on this axis.

```
child::para
```

The axis step in this example selects all `para` elements that are children of the context node.

#### **Related concepts**

“Node hierarchies” on page 5

#### **Related reference**

“Predicates” on page 63

“Abbreviated syntax for path expressions” on page 61



## Axes

An *axis* is a part of an axis step that specifies a direction of movement through an XML document.

An axis can be either a forward or reverse axis. A *forward axis* contains the context node and nodes that are after the context node in document order. A *reverse axis* contains the context node and nodes that are before the context node in document order.

The following table describes the axes that are supported in DB2 XQuery.

Table 16. Supported axes in DB2 XQuery

Axis	Description	Direction	Notes
child	Returns the children of the context node.	Forward	Document nodes and element nodes are the only nodes that have children. If the context node is any other kind of node, or if the context node is a document or element node without any children, the child axis is an empty sequence. The children of a document node or element node can be element, processing instruction, comment, or text nodes. Attribute and document nodes can never appear as children.
descendant	Returns the descendants of the context node (the children, the children of the children, and so on).	Forward	
attribute	Returns the attributes of the context node.	Forward	This axis is empty if the context node is not an element node.
self	Returns the context node only.	Forward	
descendant-or-self	Returns the context node and the descendants of the context node.	Forward	
parent	Returns the parent of the context node, or an empty sequence if the context node has no parent.	Reverse	An element node can be the parent of an attribute node even though an attribute node is never a child of an element node.

When an axis step selects a sequence of nodes, each node is assigned a context position that corresponds to its position in the sequence. If the axis is a forward axis, context positions are assigned to the nodes in document order, starting with 1. If the axis is a reverse axis, context positions are assigned to the nodes in reverse document order, starting with 1. Context position assignments allow you to select a node from the sequence by specifying its position.

### Related reference

“Node tests”

“Predicates” on page 63

## Node tests

A *node test* is a condition that must be true for each node that is selected by an axis step. The node test can be expressed as either a name test or a kind test. A *name test* selects nodes based on the name of the node. A *kind test* selects nodes based on the kind of node.

## Name tests

A name test consists of a QName or a wildcard. When a name test is specified in an axis step, the step selects the nodes on the specified axis that match the QName or wildcard. If the name test is specified on the attribute axis, then the step selects any attributes that match the name test. On all other axes, the step

selects any elements that match the name test. The QNames match if the expanded QName of the node is equal (on a codepoint basis) to the expanded QName that is specified in the name test. Two expanded QNames are equal if their namespace URIs are equal and their local names are equal (even if their namespace prefixes are not equal).

**Important:** Any prefix that is specified in a name test must correspond to one of the statically known namespaces for the expression. For name tests that are performed on the attribute axis, unprefixed QNames have no namespace URI. For name tests that are performed on all other axes, unprefixed QNames have the namespace URI of the default element/type namespace.

The following table describes the name tests that are supported in DB2 XQuery.

Table 17. Supported name tests in DB2 XQuery

Test	Description	Examples
<i>QName</i>	Matches any nodes (on the specified axis) whose QName is equal to the specified QName. If the axis is an attribute axis, this test matches attribute nodes. On all other axes, this test matches element nodes.	In the expression <code>child::para</code> , the name test <code>para</code> selects all of the <code>para</code> elements on the <code>child</code> axis.
*	Matches all nodes on the specified axis. If the axis is an attribute axis, this test matches all attribute nodes. On all other axes, this test matches all element nodes.	In the expression, <code>child::*</code> , the name test <code>*</code> matches all of the elements on the <code>child</code> axis.
<i>NCName</i> *	Specifies an <i>NCName</i> that represents the prefix part of a QName. This name test matches all nodes (on the specified axis) whose namespace URI matches the namespace URI to which the prefix is bound. If the axis is an attribute axis, this test matches attribute nodes. On all other axes, this test matches element nodes.	In the expression <code>child::ns1:*</code> , the name test <code>ns1:*</code> matches all of the elements on the <code>child</code> axis that are associated with the namespace that is bound to the prefix <code>ns1</code> .
*: <i>NCName</i>	Specifies an <i>NCName</i> that represents the local part of a QName. This name test matches any nodes (on the specified axis) whose local name is equal to the <i>NCName</i> . If the axis is an attribute axis, this test matches attribute nodes. On all other axes, this test matches element nodes.	In the expression <code>child::*:customerinfo</code> , the name test <code>*:customerinfo</code> matches all of the elements on the <code>child</code> axis that have the local name <code>customerinfo</code> , regardless of the namespace that is associated with the element name.

## Kind tests

When a kind test is specified in an axis step, the step selects only those nodes on the specified axis that match the kind test. The following table describes the kind tests that are supported in DB2 XQuery.

Table 18. Supported kind tests in DB2 XQuery

Test	Description	Examples
<code>node()</code>	Matches any node on the specified axis.	In the expression <code>child::node()</code> , the kind test <code>node()</code> selects any nodes on the <code>child</code> axis.
<code>text()</code>	Matches any text node on the specified axis.	In the expression <code>child::text()</code> , the kind test <code>text()</code> selects any text nodes on the <code>child</code> axis.
<code>comment()</code>	Matches any comment node on the specified axis.	In the expression <code>child::comment()</code> , the kind test <code>comment()</code> selects any comment nodes on the <code>child</code> axis.

Table 18. Supported kind tests in DB2 XQuery (continued)

Test	Description	Examples
processing-instruction()	Matches any processing-instruction node on the specified axis.	In the expression <code>child::processing-instruction()</code> , the kind test <code>processing-instruction()</code> selects any processing instruction nodes on the child axis.
element() or element(*)	Matches any element node on the specified axis.	In the expression <code>child::element()</code> , the kind test <code>element()</code> selects any element nodes on the child axis. In the expression <code>child::element(*)</code> , the kind test <code>element(*)</code> selects any element nodes on the child axis.
attribute() or attribute(*)	Matches any attribute node on the specified axis.	In the expression <code>child::attribute()</code> , the kind test <code>attribute()</code> selects any attribute nodes on the child axis. In the expression <code>child::attribute(*)</code> , the kind test <code>attribute(*)</code> selects any attribute nodes on the child axis.
document-node()	Matches any document node on the specified axis.	In the expression <code>self::document-node()</code> , the kind test <code>document-node()</code> selects a document node that is the context node.

#### Related reference

“Axes” on page 59

“Predicates” on page 63

## Abbreviated syntax for path expressions

XQuery provides an abbreviated syntax for expressing axes in path expressions.

The following table describes the abbreviations that are allowed in path expressions.

Table 19. Abbreviated syntax for path expressions

Abbreviated syntax	Description	Examples
no axis specified	Shorthand abbreviation for <code>child::</code> except when the axis step specifies <code>attribute()</code> for the node test. When the axis step specifies an attribute test, an omitted axis is shorthand for <code>attribute::</code> .	The path expression <code>section/para</code> is an abbreviation for <code>child::section/child::para</code> . The path expression <code>section/attribute()</code> is an abbreviation for <code>child::section/attribute::attribute()</code> .
@	Shorthand abbreviation for <code>attribute::</code> .	The path expression <code>section/@id</code> is an abbreviation for <code>child::section/attribute::id</code> .
//	Shorthand abbreviation for <code>/descendant-or-self::node()/</code> , except when this abbreviation appears at the beginning of the path expression.  When this abbreviation appears at the beginning of the path expression, the axis step selects an initial node sequence that contains the root of the tree in which the context node is found, plus all nodes that are descended from this root. This expression returns an error if the root node is not a document node.	The path expression <code>div1//para</code> is an abbreviation for <code>child::div1/descendant-or-self::node()/child::para</code> .

Table 19. Abbreviated syntax for path expressions (continued)

Abbreviated syntax	Description	Examples
..	Shorthand abbreviation for parent::node().	The path expression ../title is an abbreviation for parent::node()/child::title

## Examples of abbreviated syntax and unabbreviated syntax

The following table provides examples of abbreviated syntax and unabbreviated syntax.

Table 20. Unabbreviated syntax and abbreviated syntax

Unabbreviated syntax	Abbreviated syntax	Result
child::para	para	Selects the para elements that are children of the context node.
child::*	*	Selects all of the elements that are children of the context node.
child::text()	text()	Selects all of the text nodes that are children of the context node.
child::node()	node()	Selects all of the children of the context node. This expression returns no attribute nodes because attributes are not considered children of a node.
attribute::name	@name	Selects the name attribute of the context node
attribute::*	@*	Selects all of the attributes of the context node.
child::para[fn:position() = 1]	para[1]	Selects the first para element that is a child of the context node.
child::para[fn:position() = fn:last()]	para[fn:last()]	Selects the last para element that is a child of the context node.
/child::book/child::chapter[fn:position() = 5] /child::section[fn:position() = 2]	/book/chapter[5]/section[2]	Selects the second section of the fifth chapter of the book whose parent is the document node that contains the context node.
child::para[attribute::type="warning"]	para[@type="warning"]	Selects all para children of the context node that have a type attribute with the value warning.
child::para[attribute::type='warning'] [fn:position() = 5]	para[@type="warning"][5]	Selects the fifth para child of the context node that has a type attribute with value warning.

Table 20. Unabbreviated syntax and abbreviated syntax (continued)

Unabbreviated syntax	Abbreviated syntax	Result
<code>child::para[fn:position() = 5] [attribute::type="warning"]</code>	<code>para[5][@type="warning"]</code>	Selects the fifth <code>para</code> child of the context node if that child has a <code>type</code> attribute with value <code>warning</code> .
<code>child::chapter[child::title='Introduction']</code>	<code>chapter[title="Introduction"]</code>	Selects the chapter children of the context node that have one or more <code>title</code> children whose typed value is equal to the string <code>Introduction</code> .
<code>child::chapter[child::title]</code>	<code>chapter[title]</code>	Selects the chapter children of the context node that have one or more <code>title</code> children.

### Related reference

“Syntax of path expressions” on page 57

“Axes” on page 59

## Predicates

A *predicate* filters a sequence by retaining the qualifying items. A predicate consists of an expression, called a predicate expression, that is enclosed in square brackets (`[]`). The predicate expression is evaluated once for each item in the sequence, with the selected item as the context item. Each evaluation of the predicate expression returns an `xs:boolean` value called the *predicate truth value*. Those items for which the predicate truth value is true are retained, and those for which the predicate truth value is false are discarded.

The following rules are used to determine the predicate truth value:

- If the predicate expression returns a non-numeric value, the predicate truth value is the effective boolean value of the predicate expression.
- If the predicate expression returns a numeric value, the predicate truth value is true only for the item whose position in the sequence is equal to that numeric value. For other items, the predicate truth value is false. This kind of predicate is called a *numeric predicate* or *positional predicate*. For example, in the expression `$products[5]`, the numeric predicate `[5]` retains only the fifth item in the sequence bound to the variable `$products`.

**Important:** The item that is selected from a sequence by a numeric predicate is deterministic only if the sequence has a deterministic order.

**Tip:** The behavior of a predicate depends on whether the predicate expression returns a numeric value or not, which might not be clear from looking at the predicate expression. You can force a predicate to use an effective boolean value by using the `fn:boolean` function, as in `[fn:boolean(PredicateExpression)]`. Alternatively, you can force a predicate to behave like a positional predicate by using the `fn:position` function, as in `[fn:position() eq PredicateExpression]`.

The following examples have predicates:

- `chapter[2]` selects the second chapter element that is a child of the context node.
- `descendant::toy[@color = "Red"]` selects all of the descendants of the context node that are elements named `toy` and have a `color` attribute with the value `"Red"`.

- `employee[secretary][assistant]` selects all of the employee children of the context node that have both a secretary child element and an assistant child element.
- `(<cat />, <dog />, 47, <zebra />)[2]` returns the element `<dog />`.

**Related concepts**

“Effective Boolean value” on page 51

**Related reference**

“Order of results in XQuery expressions” on page 48

## Sequence expressions

Sequence expressions construct, filter, and combine sequences of items. Sequences are never nested. For example, combining the values 1, (2, 3), and ( ) into a single sequence results in the sequence (1, 2, 3).

**Related concepts**

“Sequences and items” on page 4

## Expressions that construct sequences

Sequences can be constructed by using either the comma operator or a range expression.

### Comma operators

To construct a sequence by using the comma operator, specify two or more operands (expressions) that are separated by commas. When the sequence expression is evaluated, the comma operator evaluates each of its operands and concatenates the resulting sequences, in order, into a single result sequence. For example, the following expression results in a sequence that contains five integers:

```
(15, 1, 3, 5, 7)
```

A sequence can contain duplicate atomic values and nodes. However, a sequence is never an item in another sequence. When a new sequence is created by concatenating two or more input sequences, the new sequence contains all of the items of the input sequences, and the length of the sequence is the sum of the lengths of the input sequences.

The following expressions use the comma operator for sequence construction:

- This expression combines four sequences of length one, two, zero, and two, respectively, into a single sequence of length five. The result of this expression is the sequence 10, 1, 2, 3, 4.  
(10, (1, 2), (), (3, 4))
- The result of this expression is a sequence that contains all salary elements that are children of the context node, followed by all bonus elements that are children of the context node.  
(salary, bonus)
- Assuming that the variable \$price is bound to the value 10.50, the result of this expression is the sequence 10.50, 10.50.  
(\$price, \$price)

### Range expressions

Range expressions construct a sequence of consecutive integers. A range expression consists of two operands (expressions) that are separated by the **to** operator. The value of each operand must be convertible to a value of type `xs:integer`. If either operand is an empty sequence, or if the integer that is derived from the first operand is greater than the integer that is derived from the second operand, the result of the range expression is an empty sequence. Otherwise, the result is a sequence that contains the two integers that are derived from the operands and every integer between the two integers, in increasing order. For example, the following range expression evaluates to the sequence 1, 2, 3, 4:

(1 to 4)

The following examples use range expressions for sequence construction:

- This example uses a range expression as one operand in constructing a sequence. The sequence expression evaluates to the sequence 10, 1, 2, 3, 4.

```
(10, 1 to 4)
```

- This example constructs a sequence of length one that contains the single integer 10.

```
10 to 10
```

- The result of this example is a sequence of length zero.

```
15 to 10
```

- This example uses the `fn:reverse` function to construct a sequence of six integers in decreasing order. This sequence expression evaluates to the sequence 15, 14, 13, 12, 11, 10.

```
fn:reverse(10 to 15)
```

### Related concepts

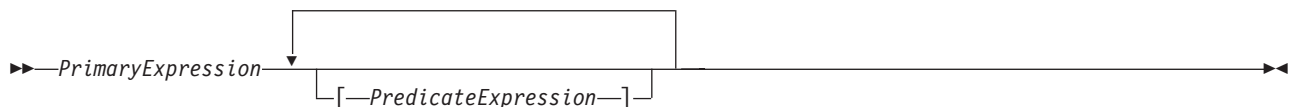
“Sequences and items” on page 4

## Filter expressions

A filter expression consists of a primary expression that is followed by zero or more predicates. The predicates, if present, filter the sequence that is returned by the primary expression.

The result of the filter expression consists of all of the items with a predicate truth value of true that are returned by the primary expression. If no predicates are specified, the result is simply the result of the primary expression. The items in the result sequence are in the same order as the items that are returned by the primary expression. During evaluation of a predicate, each item has a context position that represents its position in the sequence that is being filtered by that predicate. The first context position is 1.

## Syntax



### *PrimaryExpression*

A primary expression.

### *PredicateExpression*

An expression that determines whether items of the sequence are retained or discarded.

## Examples

The following examples use filter expressions to return a filtered sequence:

- Given a sequence of products bound to a variable, this expression returns only those products with a price that is greater than 100:

```
$products[price gt 100]
```

- This expression uses a range expression with a predicate to list all integers from 1 to 100 that are divisible by 5. The range expression is processed as a primary expression because it is enclosed in parentheses:

```
(1 to 100)[. mod 5 eq 0]
```

- This expression results in the integer 5:

```
(1 to 21)[5]
```

- This expression uses a filter expression as a step in a path expression. The expression returns the last chapter or appendix within the book that is bound to the variable \$book:

```
$book/(chapter | appendix)[fn:last()]
```

#### Related concepts

“Sequences and items” on page 4

#### Related reference

“Primary expressions” on page 52

“Predicates” on page 63

## Expressions for combining sequences of nodes

DB2 XQuery provides operators for combining sequences of nodes. These operators include **union**, **intersect**, and **except**.

The following table describes the operators that are available for combining sequences of nodes.

Table 21. XQuery operators for combining sequences of nodes

Operator	Description
<b>union</b> or	Takes two node sequences as operands and returns a sequence that contains all of the nodes that occur in either of the operands. The <b>union</b> keyword and the   character are equivalent.
<b>intersect</b>	Takes two node sequences as operands and returns a sequence that contains all of the nodes that occur in both operands.
<b>except</b>	Takes two node sequences as operands and returns a sequence that contains all of the nodes that occur in the first operand but not in the second operand.

All of these operators eliminate duplicate nodes from their result sequences based on node identity. The resulting sequence is returned in document order.

The operands of **union**, **intersect**, or **except** must resolve to sequences that contain nodes only. If an operand contains an item that is not a node, an error is returned.

In addition to the operators that are described in this topic, DB2 XQuery provides functions for indexed access to items or sub-sequences of a sequence (fn:index-of), for indexed insertion or removal of items in a sequence (fn:insert-before and fn:remove), and for removing duplicate items from a sequence (fn:distinct-values).

## Examples

In these examples, suppose that the variable \$managers is bound to a set of employee nodes that represent employees who are managers, and the variable \$students is bound to a set of employee nodes that represent employees who are students.

The following expressions are all valid examples that use operators to combine sequences of nodes:

- \$managers union \$students returns the set of nodes that represent employees who are either managers or students.
- \$managers intersect \$students returns the set of nodes that represent employees who are both managers and students.
- \$managers except \$students returns the set of nodes that represent employees who are managers but not students.

#### Related concepts



“Sequences and items” on page 4

“Node identity” on page 10

---

## Arithmetic expressions

Arithmetic expressions perform operations that involve addition, subtraction, multiplication, division, and modulus.

The following table describes the arithmetic operators and lists them in order of operator precedence from highest to lowest. Unary operators have a higher precedence than binary operators unless parentheses are used to force the evaluation of the binary operator.

Table 22. Arithmetic operators in XQuery

Operator	Purpose	Associativity
-(unary), +(unary)	negates value of operand, maintains value of operand	right-to-left
*, div, idiv, mod	multiplication, division, integer division, modulus	left-to-right
+, -	addition, subtraction	left-to-right

**Note:** A subtraction operator must be preceded by whitespace if the operator could otherwise be interpreted as part of a previous token. For example, `a-b` is interpreted as a name, but `a - b` and `a -b` are interpreted as arithmetic operations.

The result of an arithmetic expression is a numeric value, an empty sequence, or an error. When an arithmetic expression is evaluated, each operand is atomized (converted into an atomic value), and the following rules are applied:

- If the atomized operand is an empty sequence, then the result of the arithmetic expression is an empty sequence.
- If the atomized operand is a sequence that contains more than one value, an error is returned.
- If the atomized operand is an untyped atomic value (`xd:untypedAtomic`), the value is cast to `xs:double`. If the cast fails, an error is returned.

If the types of the operands, after evaluation, are a valid combination for the arithmetic operator, then the operator is applied to the atomized operands, and the result of this operation is an atomic value or an error (for example, an error might result from dividing by zero.) If the types of the operands are not a valid combination for the arithmetic operator, an error is returned.

Table 23 on page 68 identifies valid combinations of types for arithmetic operators. In this table, the letter A represents the first operand in the expression, and the letter B represents the second operand. The term numeric denotes the types `xs:integer`, `xs:decimal`, `xs:float`, `xs:double`, or any types derived from one of these types. If the result type of an operator is listed as numeric, the result type will be the first type in the ordered list (`xs:integer`, `xs:decimal`, `xs:float`, `xs:double`) into which all operands can be converted by subtype substitution and type promotion.

Table 23. Valid types for operands of arithmetic expressions

Operator with operands	Type of operand A	Type of operand B	Result type
A + B	numeric	numeric	numeric
	xs:date	xdt:yearMonthDuration	xs:date
	xdt:yearMonthDuration	xs:date	xs:date
	xs:date	xdt:dayTimeDuration	xs:date
	xdt:dayTimeDuration	xs:date	xs:date
	xs:time	xdt:dayTimeDuration	xs:time
	xdt:dayTimeDuration	xs:time	xs:time
	xs:dateTime	xdt:yearMonthDuration	xs:dateTime
	xdt:yearMonthDuration	xs:dateTime	xs:dateTime
	xs:dateTime	xdt:dayTimeDuration	xs:dateTime
	xdt:dayTimeDuration	xs:dateTime	xs:dateTime
	xdt:yearMonthDuration	xdt:yearMonthDuration	xdt:yearMonthDuration
	xdt:dayTimeDuration	xdt:dayTimeDuration	xdt:dayTimeDuration
	A - B	numeric	numeric
xs:date		xs:date	xdt:dayTimeDuration
xs:date		xdt:yearMonthDuration	xs:date
xs:date		xdt:dayTimeDuration	xs:date
xs:time		xs:time	xdt:dayTimeDuration
xs:time		xdt:dayTimeDuration	xs:time
xs:dateTime		xs:dateTime	xdt:dayTimeDuration
xs:dateTime		xdt:yearMonthDuration	xs:dateTime
xs:dateTime		xdt:dayTimeDuration	xs:dateTime
xdt:yearMonthDuration		xdt:yearMonthDuration	xdt:yearMonthDuration
xdt:dayTimeDuration		xdt:dayTimeDuration	xdt:dayTimeDuration
A * B		numeric	numeric
	xdt:yearMonthDuration	numeric	xdt:yearMonthDuration
	numeric	xdt:yearMonthDuration	xdt:yearMonthDuration
	xdt:dayTimeDuration	numeric	xdt:dayTimeDuration
	numeric	xdt:dayTimeDuration	xdt:dayTimeDuration
A idiv B	numeric	numeric	xs:integer
A div B	numeric	numeric	numeric; but xs:decimal if both operands are xs:integer
	xdt:yearMonthDuration	numeric	xdt:yearMonthDuration
	xdt:dayTimeDuration	numeric	xdt:dayTimeDuration
	xdt:yearMonthDuration	xdt:yearMonthDuration	xs:decimal
	xdt:dayTimeDuration	xdt:dayTimeDuration	xs:decimal
A mod B	numeric	numeric	numeric

## Examples

- The first expression below returns the xs:decimal value -1.5, and the second expression returns the xs:integer value -1:

```
-3 div 2  
-3 idiv 2
```

- In the following expression, the subtraction of two date values results in a value of type xdt:dayTimeDuration:

```
$emp/hiredate - $emp/birthdate
```

- The following example illustrates the difference between a subtraction operator and hyphens that are used in the variable names unit-price and unit-discount:

```
$unit-price - $unit-discount
```

### Related concepts

“Atomization” on page 50

“Whitespace” on page 14

“Subtype substitution” on page 50

“Type promotion” on page 51

Chapter 2, “Type system,” on page 17

### Related reference

“Precedence” on page 47

“Parenthesized expression” on page 55

---

## Comparison expressions

Comparison expressions compare two values. XQuery provides three kinds of comparison expressions: value comparisons, general comparisons, and node comparisons.

### Value comparisons

Value comparisons compare two atomic values. The value comparison operators include **eq**, **ne**, **lt**, **le**, **gt**, and **ge**.

The following table describes these operators.

Table 24. Value comparison operators in XQuery

Operator	Purpose
eq	Returns true if the first value is equal to the second value.
ne	Returns true if the first value is not equal to the second value.
lt	Returns true if the first value is less than the second value.
le	Returns true if the first value is less than or equal to the second value.
gt	Returns true if the first value is greater than the second value.
ge	Returns true if the first value is greater than or equal to the second value.

Two values can be compared if they have the same type or if the type of one operand is a subtype of the other operand’s type. Two operands of numeric types (types xs:float, xs:integer, xs:decimal, xs:double, and types derived from these) can be compared. Also, xs:string and xs:anyURI values can be compared.

**Special values:** For xs:float and xs:double values, positive zero and negative zero compare equal. INF equals INF, and -INF equals -INF. NaN does not equal itself. Positive infinity is greater than all other non-NaN values; negative infinity is less than all other non-NaN values. NaN **ne** NaN is true, and any other comparison involving a NaN value is false. Two

values of type `xs:QName` are considered to be equal if their namespace URIs are equal and their local names are equal (namespace prefixes are not significant).

The result of a value comparison can be a boolean value, an empty sequence, or an error. When a value comparison is evaluated, each operand is atomized (converted into an atomic value), and the following rules are applied:

- If either atomized operand is an empty sequence, then the result of the value comparison is an empty sequence.
- If either atomized operand is a sequence that contains more than one value, an error is returned.
- If either atomized operand is an untyped atomic value (`xdt:untypedAtomic`), that value is cast to `xs:string`.

Casting values of type `xdt:untypedAtomic` to `xs:string` allows value comparisons to be transitive. In contrast, general comparisons follow a different rule for casting untyped data and are therefore not transitive. The transitivity of a value comparison might be compromised by loss of precision during type conversions. For example, two `xs:integer` values that differ slightly might both be considered equal to the same `xs:float` value because `xs:float` has less precision than `xs:integer`.

- If the types of the operands, after evaluation, are a valid combination for the operator, the operator is applied to the atomized operands, and the result of the comparison is either true or false. If the types of the operands are not a valid combination for the comparison operator, an error is returned.

The following types can be compared with the **eq** or **ne** operator. The term Gregorian refers to the types `xs:gYearMonth`, `xs:gYear`, `xs:gMonthDay`, `xs:gDay`, and `xs:gMonth`. For binary operators that accept two Gregorian-type operands, both operands must have the same type (for example, if one operand is of type `xs:gDay`, the other operand must be of type `xs:gDay`). The term numeric refers to the types `xs:integer`, `xs:decimal`, `xs:float`, `xs:double`, and any type derived from one of these types. During comparisons that involve numeric values, subtype substitution and numeric type promotion are used to convert the operands into the first type in the ordered list (`xs:integer`, `xs:decimal`, `xs:float`, `xs:double`) into which all operands can be converted.

- Numeric
- `xs:boolean`
- `xs:string`
- `xs:date`
- `xs:time`
- `xs:dateTime`
- `xs:duration`
- `xdt:yearMonthDuration`
- `xdt:dayTimeDuration`
- Gregorian
- `xs:hexBinary`
- `xs:base64Binary`
- `xs:QName`
- `xs:NOTATION`

The following types can be compared with the **gt**, **lt**, **ge**, and **le** operators. The term numeric refers to the types `xs:integer`, `xs:decimal`, `xs:float`, and `xs:double`. During comparisons that involve numeric values, subtype substitution and numeric type promotion are used to convert the operands into the first type in the ordered list (`xs:integer`, `xs:decimal`, `xs:float`, `xs:double`) into which all operands can be converted.

- Numeric
- `xs:boolean`
- `xs:string`

- xs:date
- xs:time
- xs:dateTime
- xdt:yearMonthDuration
- xdt:dayTimeDuration

## Examples

- The following comparison atomizes the nodes that are returned by the expression `$book/author`. The comparison is true only if the result of atomization is the value "Kennedy" as an instance of `xs:string` or `xdt:untypedAtomic`. If the result of atomization is a sequence that contains more than one value, an error is returned

```
$book1/author eq "Kennedy"
```

- The following path expression contains a predicate that selects products whose weight is greater than 100. For any product that does not have a weight subelement, the value of the predicate is the empty sequence, and the product is not selected:

```
//product[weight gt 100]
```

- The following comparisons are true because, in each case, the two constructed nodes have the same value after atomization, even though they have different identities or names:

```
<a>5</a> eq <a>5</a>
<a>5</a> eq <b>5</b>
```

### Related concepts

"Atomic values" on page 4

Chapter 2, "Type system," on page 17

## General comparisons

General comparisons compare two sequences of any length to determine whether at least one item in the first sequence and one item in the second sequence satisfy the specified comparison. The general comparison operators include `=`, `!=`, `<`, `<=`, `>`, and `>=`.

The following table describes these operators.

Table 25. Value comparison operators in XQuery

Operator	Purpose
<code>=</code>	Returns true if some value in the first sequence is equal to some value in the second sequence.
<code>!=</code>	Returns true if some value in the first sequence is not equal to some value in the second sequence.
<code>&lt;</code>	Returns true if some value in the first sequence is less than some value in the second sequence.
<code>&lt;=</code>	Returns true if some value in the first sequence is less than or equal to some value in the second sequence.
<code>&gt;</code>	Returns true if some value in the first sequence is greater than some value in the second sequence.
<code>&gt;=</code>	Returns true if some value in the first sequence is greater than or equal to some value in the second sequence.

The result of a general comparison is either a boolean value or an error. When a general comparison is evaluated, each operand is atomized (converted into a sequence of atomic values). The first sequence is

compared to the second sequence to determine whether at least one item in the first sequence and at least one item in the second sequence satisfy the specified comparison. When comparing individual atomic values, the following rules are applied:

- If one of the atomic values is an instance of `xdt:untypedAtomic` and the other is an instance of a numeric type, then the untyped value is cast to the type `xs:double`.
- If one of the atomic values is an instance of `xdt:untypedAtomic` and the other is an instance of `xdt:untypedAtomic` or `xs:string`, then the `xdt:untypedAtomic` values are cast to the type `xs:string`.
- If one of the atomic values is an instance of `xdt:untypedAtomic` and the other is not an instance of `xs:string`, `xdt:untypedAtomic`, or any numeric type, then the `xdt:untypedAtomic` value is cast to the type of the other value.

After the types are cast as described above, the atomic values are compared using one of the value comparison operators `eq`, `ne`, `lt`, `le`, `gt`, or `ge`.

The result of the comparison is true if there is a pair of atomic values, one in the first operand sequence and the other in the second operand sequence, for which the comparison is true.

**Note:** When errors occur, the result of a general comparison can be either a boolean value or an error. For example, the comparison  $(1, 2) = (2, \text{"Hello"})$  might return true because  $2 \text{ eq } 2$  is true, or the comparison might return an error because the value 1 is not comparable with the value "Hello".

**Tip:** To compare two sequences on an item-by-item basis, use the XQuery function `fn:deep-equal`.

## Examples

- The following comparison is true if the typed value of some author subelement of `$book1` is "Kennedy" as an instance of `xs:string` or `xdt:untypedAtomic`:

```
$book1/author = "Kennedy"
```

- The following example contains three general comparisons. The value of the first two comparisons is true, and the value of the third comparison is false. This example illustrates the fact that general comparisons are not transitive:

```
(1, 2) = (2, 3)  
(2, 3) = (3, 4)  
(1, 2) = (3, 4)
```

- The following example contains two general comparisons, both of which are true. This example illustrates the fact that the `=` and `!=` operators are not inverses of each other:

```
(1, 2) = (2, 3)  
(1, 2) != (2, 3)
```

- In the following example, the variables `$a`, `$b`, and `$c` are bound to element nodes that have the type annotation `xdt:untypedAtomic`. The element nodes contain the string values "1", "2", and "2.0" respectively. In this example, the following expression returns false because the values that are bound to `$b` and `$c` ("2" and "2.0") are compared as strings:

```
($a, $b) = ($c, 3.0)
```

However, the following expression returns true because the value that is bound to `$b` ("2") and the value 2.0 are compared as numbers:

```
($a, $b) = ($c, 2.0)
```

### Related concepts

"Sequences and items" on page 4

"Atomic values" on page 4

"Atomization" on page 50

### Related reference

"deep-equal function" on page 118

## Node comparisons

Node comparisons compare two nodes. Nodes can be compared to determine if they share the same identity or if one node precedes or follows another node in document order.

The following table describes the node comparison operators that are available in XQuery.

Table 26. Node comparison operators in XQuery

Operator	Purpose
is	Returns true if the two nodes that are compared have the same identity.
<<	Returns true if the first operand node precedes the second operand node in document order.
>>	Returns true if the first operand node follows the second operand node in document order.

The result of a node comparison is either a boolean value, an empty sequence, or an error. The result of a node comparison is defined by the following rules:

- Each operand must be either a single node or an empty sequence; otherwise, an error is returned.
- If either operand is an empty sequence, the result of the comparison is an empty sequence.
- A comparison that uses the **is** operator is true when the two nodes that are compared have the same identity; otherwise, the comparison is false.
- A comparison that uses the << operator returns true when the left operand node precedes the right operand node in document order; otherwise, the comparison returns false.
- A comparison that uses the >> operator returns true when the left operand node follows the right operand node in document order; otherwise, the comparison returns false.

### Examples

- The following comparison is true only if both the left operand and right operand evaluate to exactly the same single node:

```
/books/book[isbn="1558604820"] is /books/book[call="QA76.9 C3845"]
```

- The following comparison is false because each constructed node has its own identity:

```
<a>5</a> is <a>5</a>
```

- The following comparison is true only if the node that is identified by the left operand occurs before the node that is identified by the right operand in document order:

```
/transactions/purchase[parcel="28-451"] << /transactions/sale[parcel="33-870"]
```

#### Related concepts

“Node identity” on page 10

“Document order of nodes” on page 10

“Node hierarchies” on page 5

---

## Logical expressions

Logical expressions use the operators **and** and **or** to compute a Boolean value (true or false).

The following table describes these operators and lists them in order of operator precedence from highest to lowest.

Table 27. Logical expression operators in XQuery

Operator	Purpose
and	Returns true if both expressions are true.
or	Returns true if one or both expressions are true.

The result of a logical expression is either a Boolean value (true or false) or an error. When a logical expression is evaluated, the effective Boolean value (EBV) of each operand is determined. The operator is then applied to the EBVs of the operands, and the result is either a boolean value or an error. If the EBV of an operand is an error, then the logical expression might result in an error. The following table shows the results that are returned by a logical expression based on the EBVs of its operands.

Table 28. Results of logical expressions based on EBVs of operands

EBV of operand 1	Operator	EBV of operand 2	Result
true	and	true	true
true	and	false	false
false	and	true	false
false	and	false	false
true	and	error	error
error	and	true	error
false	and	error	false or error
error	and	false	false or error
error	and	error	error
true	or	true	true
false	or	false	false
true	or	false	true
false	or	true	true
true	or	error	true or error
error	or	true	true or error
false	or	error	error
error	or	false	error
error	or	error	error

**Tip:** In addition to logical expressions, XQuery provides a function named `fn:not` that takes a general sequence as a parameter and returns a Boolean value.

## Examples

- The following expressions return true:
  - `1 eq 1 and 2 eq 2`
  - `1 eq 1 or 2 eq 3`
- The following expression might return either false or an error:
  - `1 eq 2 and 3 idiv 0 = 1`
- The following expression might return either true or an error:
  - `1 eq 1 or 3 idiv 0 = 1`
- The following expression returns an error:
  - `1 eq 1 and 3 idiv 0 = 1`

### Related concepts

“Effective Boolean value” on page 51

### Related reference

“not function” on page 137



---

## Constructors

Constructors create XML structures within a query. XQuery provides constructors for creating element nodes, attribute nodes, document nodes, text nodes, processing instruction nodes, and comment nodes. XQuery provides two kinds of constructors: direct constructors and computed constructors.

*Direct constructors* use an XML-like notation to create XML structures within a query. XQuery provides direct constructors for creating element nodes (which might include attribute nodes, text nodes, and nested element nodes), processing instruction nodes, and comment nodes. For example, the following constructor creates a book element that contains an attribute and some nested elements:

```
<book isbn="isbn-0060229357">
  <title>Harold and the Purple Crayon</title>
  <author>
    <first>Crockett</first>
    <last>Johnson</last>
  </author>
</book>
```

*Computed constructors* use a notation that is based on enclosed expressions to create XML structures within a query. A computed constructor begins with a keyword that identifies the type of node to be created and is followed by the name of the node, if applicable, and an enclosed expression that computes the content of the node. XQuery provides computed constructors for creating element nodes, attribute nodes, document nodes, text nodes, processing-instruction nodes, and comment nodes. For example, the following query contains computed constructors that generate the same result as the direct constructor described above:

```
element book {
  attribute isbn { "isbn-0060229357" },
  element title { "Harold and the Purple Crayon" },
  element author {
    element first { "Crockett" },
    element last { "Johnson" }
  }
}
```

### Related concepts

“Node kinds” on page 7

## Enclosed expressions in constructors

Enclosed expressions are used in constructors to provide computed values for element and attribute content. These expressions are evaluated and replaced by their value when the constructor is processed. Enclosed expressions are enclosed in curly braces ({} ) to distinguish them from literal text.

Enclosed expressions can be used in the following constructors to provide computed values:

- Direct element constructors:
  - An attribute value in the start tag of a direct element constructor can include an enclosed expression.
  - The content of a direct element constructor can include an enclosed expression that computes both the content and the attributes of the constructed node.
- Computed constructors:
  - An enclosed expression can be used to generate the content of the node.

For example, the following direct element constructor includes an enclosed expression:

```
<example>
  <p> Here is a query. </p>
  <eg> $b/title </eg>
  <p> Here is the result of the query. </p>
  <eg>{ $b/title }</eg>
</example>
```

When this constructor is evaluated, it might produce the following result (whitespace is added to this example to improve readability):

```
<example>
  <p> Here is a query. </p>
  <eg> $b/title </eg>
  <p> Here is the result of the query. </p>
  <eg><title>Harold and the Purple Crayon</title></eg>
</example>
```

**Tip:** To use a curly brace as an ordinary character within the content of an element or attribute, you can either include a pair of identical curly braces or use character references. For example, you can use the pair {{ to represent the character {. Likewise, you can use the pair }} to represent }. Alternatively, you can use the character references &#x7b; and &#x7d; to denote curly brace characters. A single left curly brace ({) is interpreted as the beginning delimiter for an enclosed expression. A single right curly brace (}) without a matching left curly brace is an error.

**Related reference**

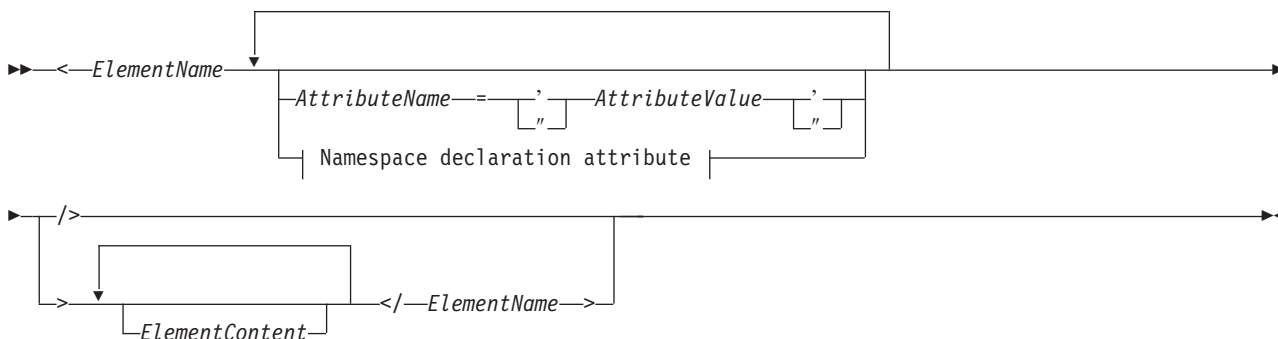
- “Computed attribute constructors” on page 84
- “Document node constructors” on page 85
- “Text node constructors” on page 86
- “Computed processing instruction constructors” on page 87
- “Computed comment constructors” on page 88

## Direct element constructors

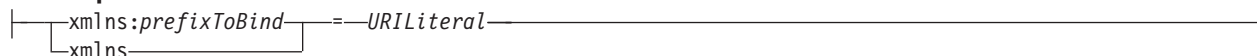
Direct element constructors use an XML-like notation to create element nodes. The constructed node can be a simple element or a complex element that contains attributes, text content, and nested elements.

The result of a direct element constructor is a new element node that has its own node identity. All of the attribute and descendant nodes of the new element node are also new nodes that have their own identities.

### Syntax



**Namespace declaration attribute:**



**ElementName**

A QName that represents the name of the element to construct. The name that is used for *ElementName* in the end tag must exactly match the name that is used in the corresponding start tag, including the prefix or absence of a prefix. If *ElementName* includes a namespace prefix, the prefix is

resolved to a namespace URI by using the statically known namespaces. If *ElementName* has no namespace prefix, the name is implicitly qualified by the default element/type namespace. The expanded QName that results from evaluating *ElementName* becomes the name of the constructed element node.

#### *AttributeName*

A QName that represents the name of the attribute to construct. If *AttributeName* includes a namespace prefix, the prefix is resolved to a namespace URI by using the statically known namespaces. If *AttributeName* has no namespace prefix, the attribute is in no namespace. The expanded QName that results from evaluating *AttributeName* becomes the name of the constructed attribute node. The expanded QName of each attribute must be unique, or the expression results in an error.

Each attribute in a direct element constructor creates a new attribute node, with its own node identity. The parent of the new attribute node is the constructed element node. The new attribute node is given a type annotation of `xdt:untypedAtomic`.

#### *AttributeValue*

A string of characters that specify a value for the attribute. The attribute value can contain enclosed expressions (expressions that are enclosed in curly braces) that are evaluated and replaced by their value when the element constructor is processed. Predefined entity references and character references are also valid and get replaced by the characters that they represent. The following table lists special characters that are valid within *AttributeValue*, but must be represented by double characters or an entity reference.

Table 29. Representation of special characters in attribute values

Character	Representation required in attribute values
{	two open curly braces ({{)
}	two closed curly braces (}}
<	&lt;
&	&amp;
"	&quot; or two double quotation marks ("" )
'	&apos; or two single quotation marks ("" )

### **xmlns**

The word that begins a namespace declaration attribute. When specified as a prefix in a QName, **xmlns** indicates that the value of *prefixToBind* will be bound to the URI that is specified by *URILiteral*. This namespace binding is added to the statically-known namespaces for this constructor expression and for all of the expressions that are nested inside of the expression (unless the binding is overridden by a nested namespace declaration attribute). In the following example, the namespace declaration attribute `xmlns:metric = "http://example.org/metric/units"` binds the prefix `metric` to the namespace `http://example.org/metric/units`.

When specified as the complete QName with no prefix, **xmlns** indicates that the default element/type namespace is set to the value of *URILiteral*. This default element/type namespace is in effect for this constructor expression and for all expressions that are nested inside of the constructor expression (unless the declaration is overridden by a nested namespace declaration attribute). In the following example, the namespace declaration attribute `xmlns = "http://example.org/animals"` sets the default element/type namespace to `http://example.org/animals`.

#### *prefixToBind*

The prefix to be bound to the URI that is specified for *URILiteral*. The value of *prefixToBind* cannot be `xml` or `xmlns`. Specifying either of these values results in an error.

#### *URILiteral*

A string literal (a sequence of zero or more characters that is enclosed in single quotation marks or double quotation marks) that represents a URI. The string literal value must be a valid URI. The value

of *URILiteral* can be a zero-length string only when the namespace declaration attribute is being used to set the default element/type namespace. Otherwise, specifying a zero-length string for *URILiteral* results in an error.

### *ElementContent*

The content of the direct element constructor. The content consists of everything between the start tag and end tag of the constructor. How boundary space is handled within element constructors is controlled by the boundary-space declaration in the prolog. The resulting content sequence is a concatenation of the content entities. Any resulting adjacent text characters, including text resulting from enclosed expressions, are merged into a single text node. Any resulting attribute nodes must come before any other content in the resulting content sequence.

*ElementContent* can consist of any of the following content:

- **Text characters.** Text characters create text nodes and adjacent text nodes are merged into a single text node. Line endings within sequences of characters are normalized according to the rules for end-of-line handling that are specified for XML 1.0. The following table lists special characters that are valid within *ElementContent*, but must be represented by double characters or an entity reference.

Table 30. Representation of special characters in element content

Character	Representation required in element content
{	two open curly braces ({{})
}	two closed curly braces (}})
<	&lt;
&	&amp;

- **Nested direct constructors.**
- **CDataSections.** CDataSections are specified using the following syntax: `<![CDATA[contents]]>` where *contents* consists of a series of characters. The characters that are specified for *contents*, including special characters such as < and &, are treated as literal characters rather than as delimiters. The sequence `]]>` terminates the CDataSection and is therefore not allowed within *contents*.
- **Character references and predefined entity references.** During processing, predefined entity references and character references are expanded into their referenced strings.
- **Enclosed expressions.** An enclosed expression is an XQuery expression that is enclosed in curly braces. For example, `{5 + 7}` is an enclosed expression. The value of an enclosed expression can be any sequence of nodes and atomic values. Enclosed expressions can be used within the content of a direct element constructor to compute both the content and the attributes of the constructed node. For each node that is returned by an enclosed expression, a new copy is made of the node and all of its descendants, which retain their original type annotations. Any attribute nodes that are returned by *ElementContent* must be at the beginning of the resulting content sequence; these attribute nodes become attributes of the constructed element. Any element, content, or processing instruction nodes that are returned by *ElementContent* become children of the newly constructed node. Any atomic values that are returned by *ElementContent* are converted to strings and stored in text nodes, which become children of the constructed node. Adjacent text nodes are merged into a single text node.

## Examples

- The following direct element constructor creates a book element. The book element contains complex content that includes an attribute node, some nested element nodes, and some text nodes:

```
<book isbn="isbn-0060229357">
  <title>Harold and the Purple Crayon</title>
  <author>
```

```

    <first>Crockett</first>
    <last>Johnson</last>
  </author>
</book>

```

- The following examples demonstrate how element content is processed in direct element constructors:

- The following expression constructs an element node that has one child, a text node that contains the value "1":

```
<a>{1}</a>
```

- The following expression constructs an element node that has one child, a text node that contains the value "1 2 3":

```
<a>{1, 2, 3}</a>
```

- The following expression constructs an element node that has one child, a text node that contains the value "123":

```
<c>{1}{2}{3}</c>
```

- The following expression constructs an element node that has one child, a text node that contains the value "1 2 3":

```
<b>{1, "2", "3"}</b>
```

- The following expression constructs an element node that has one child, a text node that contains the value "I saw 8 cats.":

```
<fact>I saw 8 cats.</fact>
```

- The following expression constructs an element node that has one child, a text node that contains the value "I saw 8 cats."

```
<fact>I saw {5 + 3} cats.</fact>
```

- The following expression constructs an element node that has three children: a text node that contains "I saw ", a child element node that is named `howmany`, and a text node that contains " cats." The child element node has a single child, a text node that contains the value "8".

```
<fact>I saw <howmany>{5 + 3}</howmany> cats.</fact>
```

#### Related reference

"Predefined entity references" on page 53

"Character references" on page 54

"Enclosed expressions in constructors" on page 75

"Namespace declaration attributes"

"Boundary-space declaration" on page 40

"Boundary whitespace in direct element constructors" on page 80

## Namespace declaration attributes

Namespace declaration attributes are specified in the start tag of a direct element constructor. Namespace declaration attributes are used for two purposes: to bind a namespace prefix to a URI, and to set the default element/type namespace for the constructed element node and for its attributes and descendants.

Syntactically, a namespace declaration attribute has the same form as an attribute in a direct element constructor: the attribute is specified by a name and a value. The attribute name is constant QName. The attribute value is a string literal that represents a valid URI.

A namespace declaration attribute does not cause an attribute node to be created.

**Important:** The name of each namespace declaration attribute in a direct element constructor must be unique, or the expression results in an error.

## Binding a namespace prefix to a URI

If the QName begins with the prefix `xmlns` followed by a local name, then the declaration is used to bind the namespace prefix (specified as the local name) to a URI (specified as the attribute value). For example, the namespace declaration attribute `xmlns:metric = "http://example.org/metric/units"` binds the prefix `metric` to the namespace `http://example.org/metric/units`.

When the namespace declaration attribute is processed, the prefix and URI are added to the statically known namespaces of the constructor expression, and the new binding overrides any existing binding of the given prefix. The prefix and URI are also added as a namespace binding to the in-scope namespaces of the constructed element.

For example, in the following element constructor, namespace declaration attributes are used to bind the namespace prefixes `metric` and `english`:

```
<box xmlns:metric = "http://example.org/metric/units"
      xmlns:english = "http://example.org/english/units">
  <height> <metric:meters>3</metric:meters> </height>
  <width> <english:feet>6</english:feet> </width>
  <depth> <english:inches>18</english:inches> </depth>
</box>
```

## Setting the default element/type namespace

If the QName is `xmlns` with no prefix, then the declaration is used to set the default element/type namespace. For example, the namespace declaration attribute `xmlns = "http://example.org/animals"` sets the default element/type namespace to `http://example.org/animals`.

When the namespace declaration attribute is processed, the value of the attribute is interpreted as a namespace URI. This URI specifies the default element/type namespace of the constructor expression, and the new specification overrides any existing default. The URI is also added (with no prefix) to the in-scope namespaces of the constructed element, and the new specification overrides any existing namespace binding that has no prefix. If the namespace URI is a zero-length string, then the default element/type namespace of the constructor expression is set to "none".

For example, in the following direct element constructor, a namespace declaration attribute sets the default element/type namespace to `http://example.org/animals`:

```
<cat xmlns = "http://example.org/animals">
  <breed>Persian</breed>
</cat>
```

### Related concepts

"XML namespaces and QNames" on page 12

### Related reference

"In-scope namespaces of a constructed element" on page 82

## Boundary whitespace in direct element constructors

Within a direct element constructor, *boundary whitespace* is a sequence of consecutive whitespace characters that is delimited at each end either by the start or end of the content, or by a direct constructor, or by an enclosed expression. For example, boundary whitespace might be used in the content of the constructor to separate the end tag of a direct constructor from the start tag of a nested element.

The following diagram shows an example of a direct element constructor, with the boundary whitespace highlighted:

```

<product>
  <description> { " enclosed expression " }
</description>
</product>

```

The boundary whitespace in this example includes the following characters: a newline character and four space characters that occur between the start tags of the product and description elements; four space characters that occur between the start tag of the description element and the enclosed expression; four space characters that occur between the enclosed expression and the end tag of the description element; and one newline character that appears after the end tag of the description element.

Boundary whitespace does not include any of the following types of whitespace:

- Whitespace that is generated by an enclosed expression
- Characters that are generated by character references (for example, `&#x20;`) or by CDataSections
- Whitespace characters that are adjacent to a character reference or a CDataSection

The boundary-space policy controls whether boundary whitespace is preserved by element constructors. This policy is specified by a boundary-space declaration in the query prolog. If the boundary-space declaration specifies **strip**, then boundary whitespace is discarded. If the boundary-space declaration specifies **preserve**, then boundary whitespace is preserved. If no boundary-space declaration is specified, then the default behavior is to strip boundary whitespace during element construction.

### Examples

- In the following example, the constructed cat element node has two child element nodes that are named breed and color:

```

<cat>
  <breed>{$b}</breed>
  <color>{$c}</color>
</cat>

```

Because the boundary-space policy is **strip** by default, the whitespace that surrounds the child elements will be stripped away by the element constructor.

- In the following example, the boundary-space policy is **strip**. This example is equivalent to `<a>abc</a>`:

```

declare boundary-space strip;
<a> {"abc"} </a>

```

- In the following example, however, the boundary-space policy is **preserve**. This example is equivalent to `<a> abc </a>`:

```

declare boundary-space preserve;
<a> {"abc"} </a>

```

Because the boundary-space policy is **preserve**, the spaces that appear before and after the enclosed expression will be preserved by the element constructor.

- In the following example, the whitespace that surrounds the z is not boundary whitespace. The whitespace is always preserved, and this example is equivalent to `<a> z abc</a>`:

```

<a> z {"abc"}</a>

```

- In the following example, the whitespace characters that are generated by the character reference and adjacent whitespace characters are preserved, regardless of the boundary-space policy. This example is equivalent to `<a>     abc</a>`:

```

<a>     &#x20; {"abc"}</a>

```



- In the following example, the whitespace in the enclosed expression is preserved, regardless of the boundary-space policy, because whitespace that is generated by an enclosed expression is never considered to be boundary whitespace. This example is equivalent to `<a> </a>`:

```
<a>{"  "}</a>
```

The two spaces in the enclosed expression will be preserved by the element constructor and will appear between the start tag and the end tag in the result.

#### Related concepts

“Whitespace” on page 14

#### Related reference

“Boundary-space declaration” on page 40

“Direct element constructors” on page 76

## In-scope namespaces of a constructed element

A constructed element node has an `in-scope namespaces` property that consists of a set of namespace bindings. Each namespace binding associates a namespace prefix with a URI. The namespace bindings define the set of namespace prefixes that are available for interpreting QNames within the scope of an element.

**Important:** To understand this topic, you need to understand the difference between the following concepts:

#### Statically known namespaces

*Statically known namespaces* is a property of an expression. This property denotes the set of namespace bindings that are used by XQuery to resolve namespace prefixes during the processing of the expression. These bindings are not part of the query result.

#### In-scope namespaces

*In-scope namespaces* is a property of an element node. This property denotes the set of namespace bindings that are available to applications outside of XQuery when the element and its content are processed. These bindings are serialized as part of the query result so they will be available to outside applications.

The in-scope namespaces of a constructed element include all of the namespace bindings that are created in the following ways:

- **Explicitly through namespace declaration attributes.** A namespace binding is created for each namespace declaration attribute that is declared in the following constructors:
  - The current element constructor.
  - An enclosing direct element constructor (unless the namespace declaration attribute is overridden by the current element constructor or an intermediate constructor).
- **Automatically by the system.** A namespace binding is created in the following situations:
  - To bind the prefix `xml` to the namespace URI `http://www.w3.org/XML/1998/namespace`. This binding is created for every constructed element.
  - For each namespace that is used in the name of a constructed element or in the names of its attributes (unless the namespace binding already exists in the in-scope namespaces of the element). If the name of the node includes a prefix, then the prefix is used in the namespace binding. If the name has no prefix, then a binding is created for the empty prefix. If a conflict arises that would require two different bindings of the same prefix, then the prefix that is used in the node name is changed to an arbitrary prefix, and the namespace binding is created for the arbitrary prefix.

**Remember:** A prefix that is used in a QName must resolve to a valid URI, or a binding for that prefix cannot be added to the in-scope namespaces of the element. If the QName cannot be resolved, the expression results in an error.



## Examples

The following query includes a prolog that contains namespace declarations and a body that contains a direct element constructor:

```
declare namespace p="http://example.com/ns/p";
declare namespace q="http://example.com/ns/q";
declare namespace f="http://example.com/ns/f";

<p:newElement q:b="{f:func(2)}" xmlns:r="http://example.com/ns/r"/>
```

The namespace declarations in the prolog add the namespace bindings to the statically known namespaces of the expression. However, the namespace bindings are added to the in-scope namespaces of the constructed element only if the QNames in the constructor use these namespaces. Therefore, the in-scope namespaces of `p:newElement` consist of the following namespace bindings:

- `p = "http://example.com/ns/p"` - This namespace binding is added to the in-scope namespaces because the prefix `p` appears in the QName `p:newElement`.
- `q = "http://example.com/ns/q"` - This namespace binding is added to the in-scope namespaces because the prefix `q` appears in the attribute QName `q:b`.
- `r = "http://example.com/ns/r"` - This namespace binding is added to the in-scope namespaces because it is defined by a namespace declaration attribute.
- `xml = "http://www.w3.org/XML/1998/namespace"` - This namespace binding is added to the in-scope namespaces because it is defined for every constructed element node.

Notice that no binding for the namespace `f="http://example.com/ns/f"` is added to the in-scope namespaces. This is because the element constructor does not include element or attribute names that use the prefix `f` (even though `f:func(2)` appears in the content of the attribute named `q:b`). Therefore, this namespace binding does not appear in the query result, even though it is present in the statically known namespaces and is available for use during processing of the query.

### Related concepts

“XML namespaces and QNames” on page 12

### Related reference

“Namespace declaration attributes” on page 79

“Computed element constructors”

## Computed element constructors

A computed element constructor creates an element node for which the content of the node is computed based on an enclosed expression.

The result of a computed element constructor is a new element node that has its own node identity. All of the attribute and descendant nodes of the new element node are also new nodes that have their own identities, even if they are copies of existing nodes.

## Syntax

▶▶ element — *ElementName* — { *ContentExpression* } ▶▶

### element

A keyword that indicates that an element node will be constructed.

### *ElementName*

The QName of the element to construct. If *ElementName* includes a namespace prefix, the prefix is resolved to a namespace URI by using the statically known namespaces. If *ElementName* has no

namespace prefix, the name is implicitly qualified by the default element/type namespace. The expanded QName that results from evaluating *ElementName* becomes the name of the constructed element node.

### *ContentExpression*

An expression that generates the content of the constructed element node. The value of *ContentExpression* can be any sequence of nodes and atomic values. *ContentExpression* can be used to compute both the content and the attributes of the constructed node. For each node that is returned by *ContentExpression*, a new copy is made of the node and all of its descendants, which retain their original type annotations. Any attribute nodes that are returned by *ContentExpression* must be at the beginning of the node sequence (before any other nodes); these attribute nodes become attributes of the constructed element. Any element, content, or processing instruction nodes that are returned by *ContentExpression* become children of the newly constructed node. Any atomic values that are returned by *ContentExpression* are converted to strings and stored in text nodes, which become children of the constructed node. Adjacent text nodes are merged into a single text node.

## Examples

In the following expression, a computed element constructor makes a modified copy of an existing element. Suppose that the variable *\$e* is bound to an element that has numeric content. This constructor creates a new element named `length` that has the same attributes as *\$e* and has numeric content equal to twice the content of *\$e*:

```
element length {$e/@*, 2 * fn:data($e)}
```

In this example, if the variable *\$e* is bound to the expression `let $e := <length units="inches">{5}</length>`, then the result of the example expression is the element `<length units="inches">10</length>`.

### Related reference

“Enclosed expressions in constructors” on page 75

“In-scope namespaces of a constructed element” on page 82

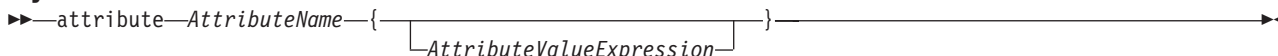
## Computed attribute constructors

A computed attribute constructor creates an attribute node for which the attribute value is computed based on an enclosed expression.

The result of a computed attribute constructor is a new attribute node that has its own node identity.

**Note:** To construct an attribute node directly, declare the attribute in a direct element constructor.

## Syntax



The diagram shows the syntax for an attribute constructor: `attribute-AttributeName-{AttributeValueExpression}`. It features a horizontal line with arrowheads at both ends. The text `attribute-AttributeName-` is positioned above the line, and `{AttributeValueExpression}` is positioned below the line. A bracket connects the start of the line to the start of the `{AttributeValueExpression}` block.

### attribute

A keyword that indicates that an attribute node will be constructed.

### *AttributeName*

The QName of the attribute to construct. If *AttributeName* includes a namespace prefix, the prefix is resolved to a namespace URI by using the statically known namespaces. If *AttributeName* has no namespace prefix, the attribute is in no namespace. The expanded QName that results from evaluating *AttributeName* becomes the name of the constructed attribute node. The expanded QName of each attribute in an element must be unique, or the expression results in an error.

### *AttributeValueExpression*

An expression that generates the value of the attribute node. During processing, atomization is applied to the result of *AttributeValueExpression*, and each atomic value in the resulting sequence is cast to a

string. The individual strings that result from the cast are concatenated with an intervening space character. The concatenated string becomes the value of the constructed attribute node.

## Example

The following computed attribute constructor constructs an attribute named `size` with a value of "7".

```
attribute size {4 + 3}
```

### Related concepts

"Attribute nodes" on page 8

### Related reference

"Enclosed expressions in constructors" on page 75

"Direct element constructors" on page 76

## Document node constructors

All document node constructors are computed constructors. A document node constructor creates a document node for which the content of the node is computed based on an enclosed expression. A document node constructor is useful when the result of a query is a complete document.

The result of a document node constructor is a new document node that has its own node identity.

**Important:** No validation is performed on the constructed document node. The XQuery document node constructor does not enforce the XML 1.0 rules that govern the structure of an XML document. For example, a document node is not required to have exactly one child that is an element node.

## Syntax

▶▶—document—{—*ContentExpression*—}—▶▶

### document

A keyword that indicates that a document node will be constructed.

### *ContentExpression*

An expression that generates the content of the constructed document node. The value of *ContentExpression* can be any sequence of nodes and atomic values except for an attribute node. Attribute nodes in the content sequence result in an error. Document nodes in the content sequence are replaced by their children. For each node that is returned by *ContentExpression*, a new copy is made of the node and all of its descendants, which retain their original type annotations. Any atomic values that are returned by the content expression are converted to strings and stored in text nodes, which become children of the constructed document node. Adjacent text nodes are merged into a single text node.

## Examples

The following document node constructor includes a content expression that returns an XML document that contains a root element named `customer-list`:

```
document
{
  <customer-list>
    {db2-fn:xmlcolumn('MYSCHEMA.CUSTOMER.INFO')/ns1:customerinfo/name}
  </customer-list>
}
```

### Related concepts

"Document nodes" on page 7

### Related reference

“Enclosed expressions in constructors” on page 75

## Text node constructors

All text node constructors are computed constructors. A text node constructor creates a text node for which the content of the node is computed based on an enclosed expression.

The result of a text node constructor is a new text node that has its own node identity.

### Syntax

►►text{—*ContentExpression*—}—

#### text

A keyword that indicates that a text node will be constructed.

#### *ContentExpression*

An expression that generates the content of the constructed text node. During processing, atomization is applied to the result of *ContentExpression*, and each atomic value in the resulting sequence is cast to a string. The individual strings that result from the cast are concatenated with an intervening space character. The concatenated string becomes the content of the constructed text node. If atomization results in an empty sequence, no text node is constructed.

**Note:** A text node constructor can be used to construct a text node that contains a zero-length string. However, if this text node is used in the content of a constructed element or a document node, then the text node is deleted or merged with another text node.

### Example

The following constructor creates a text node that contains the string "Hello":

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

#### Related concepts

“Text nodes” on page 9

#### Related reference

“Enclosed expressions in constructors” on page 75

## Processing instruction constructors

Processing instruction constructors create processing instruction nodes. XQuery provides both direct and computed constructors for creating processing instruction nodes.

The constructed node has the following node properties:

#### A target property

Identifies the application to which the processing instruction is directed.

#### A content property

Specifies the content of the processing instruction.

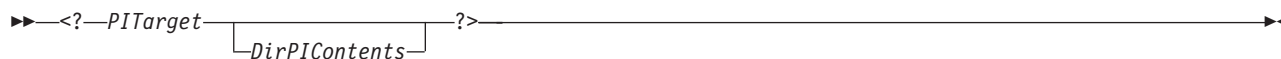
#### Related concepts

“Processing instruction nodes” on page 9

## Direct processing instruction constructors

Direct processing instruction constructors use an XML-like notation to create processing instruction nodes.

### Syntax



### *PITarget*

An NCName that represents the name of the processing application to which the processing instruction is directed. The PI target of a processing instruction cannot consist of the characters "XML" in any combination of uppercase and lowercase.

### *DirPIContents*

A series of characters that specify the contents of the processing instruction. The contents of a processing instruction cannot contain the string ?>.

## Example

The following constructor creates a processing instruction node:

```
<?format role="output" ?>
```

### Related concepts

"Processing instruction nodes" on page 9

### Related reference

"Computed processing instruction constructors"

## Computed processing instruction constructors

A computed processing instruction constructor creates a processing instruction node for which the content of the node is computed based on an enclosed expression.

The result of a computed processing instruction constructor is a new processing instruction node that has its own node identity.

## Syntax



### processing-instruction

A keyword that indicates that a processing instruction node will be constructed.

### *PITarget*

An NCName that represents the name of the processing application to which the processing instruction is directed. This name must conform to the format for NCNames that is specified by *Namespaces in XML*.

### *PIContentExpression*

An expression that generates the content of the processing instruction node. During processing, atomization is applied to the result of *PIContentExpression*, and each atomic value in the resulting sequence is cast to a string. The individual strings that result from the cast are concatenated with an intervening space character. Leading whitespace is removed, and the concatenated string becomes the content of the processing instruction node. If atomization results in an empty sequence, the sequence is replaced by a zero-length string. The content sequence cannot contain the string "?>".

## Example

The following computed constructor creates the processing instruction <?audio-output beep?>:

```
processing-instruction audio-output {"beep"}
```

### Related concepts

"Processing instruction nodes" on page 9

### Related reference

“Direct processing instruction constructors” on page 86

“Enclosed expressions in constructors” on page 75

## Comment constructors

Comment constructors create comment nodes. XQuery provides both direct and computed constructors for creating comment nodes.

### Related concepts

“Comment nodes” on page 9

“Comments” on page 15

## Direct comment constructors

Direct comment constructors use an XML-like notation to create comment nodes.

### Syntax

```
►<!-- DirCommentContents --->◄
```

#### *DirCommentContents*

A series of characters that specify the contents of the comment. The contents of a comment cannot contain two consecutive hyphens or end with a hyphen.

### Examples

The following constructor creates a comment node:

```
<!-- This is an XML comment. -->
```

### Related concepts

“Comment nodes” on page 9

### Related reference

“Computed comment constructors”

## Computed comment constructors

A computed comment constructor creates a comment node for which the content of the node is computed based on an enclosed expression.

The result of a computed comment constructor is a new comment node that has its own node identity.

### Syntax

```
►<comment—{CommentContents—}<—>◄
```

#### **comment**

A keyword that indicates that a comment node will be constructed.

#### *CommentContents*

An expression that generates the content of the comment. During processing, atomization is applied to the result of *CommentContents*, and each atomic value in the atomized sequence is cast to a string. The individual strings that result from the cast are concatenated with an intervening space character. If atomization results in an empty sequence, the sequence is replaced by a zero-length string. The content sequence cannot contain two adjacent hyphens or end with a hyphen.

### Examples

The following computed constructor creates the comment `<!--Houston, we have a problem.-->` :

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

### Related concepts

“Comment nodes” on page 9

### Related reference

“Direct comment constructors” on page 88

“Enclosed expressions in constructors” on page 75

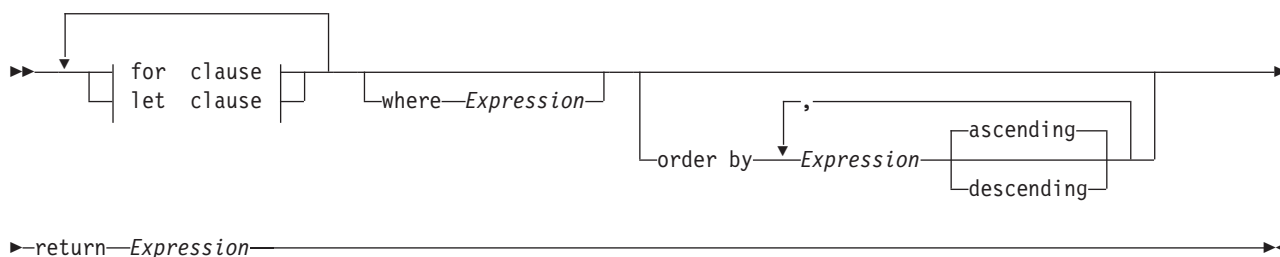
---

## FLWOR expressions

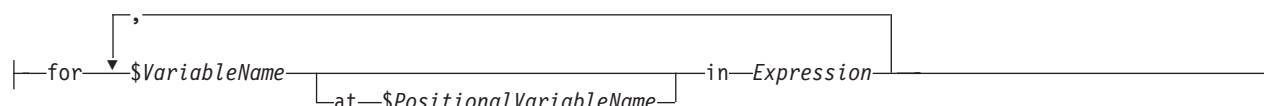
FLWOR expressions iterate over sequences and bind variables to intermediate results. FLWOR expressions are useful for computing joins between two or more documents, restructuring data, and sorting the result.

### Syntax of FLWOR expressions

A FLWOR expression is composed of the following clauses, some of which are optional: **for**, **let**, **where**, **order by**, and **return**.



#### for clause:



#### let clause:



#### for

The keyword that begins a **for** clause. A **for** clause iterates over the result of *Expression* and binds *VariableName* to each item that is returned by *Expression*.

**let** The keyword that begins a **let** clause. A **let** clause binds *VariableName* to the entire result of *Expression*.

#### *VariableName*

The name of the variable to bind to the result of *Expression*.

#### *PositionalVariableName*

The name of an optional variable that is bound to the position within the input stream of the item that is bound by each iteration of the **for** clause.

#### *Expression*

Any XQuery expression. If the expression includes a top-level comma operator, then the expression must be enclosed in parentheses.

**where**

The keyword that begins a **where** clause. A **where** clause filters the tuples of variable bindings that are generated by the **for** and **let** clauses.

**order by**

The keywords that begin an **order by** clause. An **order by** clause specifies the order in which values are processed by the **return** clause.

**ascending**

Specifies that ordering keys are processed in ascending order.

**descending**

Specifies that ordering keys are processed in descending order.

**return**

The keyword that begins a **return** clause. The expression in the **return** clause is evaluated once for each tuple of bound variables that is generated by the **for**, **let**, **where**, and **order by** clauses. The results of all of the evaluations of the return clause are concatenated into a single sequence, which is the result of the FLWOR expression.

**Related reference**

“**for** and **let** clauses”

“**where** clauses” on page 94

“**order by** clauses” on page 95

“**return** clauses” on page 97

“Variable references” on page 54

“FLWOR examples” on page 97

## for and let clauses

A **for** or **let** clause in a FLWOR expression binds one or more variables to values that will be used in other clauses of the FLWOR expression.

**Related reference**

“Syntax of FLWOR expressions” on page 89

“**order by** clauses” on page 95

## for clauses

A **for** clause iterates through the result of an expression and binds a variable to each item in the sequence.

The simplest type of **for** clause contains one variable and an associated expression. In the following example, the **for** clause includes a variable called `$i` and an expression that constructs the sequence (1, 2, 3):

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

When the **for** clause is evaluated, three variable bindings are created (one binding for each item in the sequence):

```
$i = 1
$i = 2
$i = 3
```

The **return** clause in the example executes once for each binding. The expression results in the following output:

```
<output>1</output>
<output>2</output>
<output>3</output>
```



A **for** clause can contain multiple variables, each of which is bound to the result of an expression. In the following example, a **for** clause contains two variables, `$a` and `$b`, and expressions that construct the sequences 1 2 and 4 5:

```
for $a in (1, 2), $b in (4, 5)
return <output>{$a, $b}</output>
```

When the **for** clause is evaluated, a tuple of variable bindings is created for each combination of values. This results in four tuples of variable bindings:

```
($a = 1, $b = 4)
($a = 2, $b = 4)
($a = 1, $b = 5)
($a = 2, $b = 5)
```

The **return** clause in the example executes once for each tuple of bindings. The expression results in the following output:

```
<output>1 4</output>
<output>2 4</output>
<output>1 5</output>
<output>2 5</output>
```

When the binding expression evaluates to an empty sequence, no **for** binding is generated, and no iteration is performed.

## Positional variables in for clauses

Each variable that is bound in a **for** clause can have an associated positional variable that is bound at the same time. The name of the positional variable is preceded by the keyword **at**. When a variable iterates over the items in a sequence, the positional variable iterates over the integers that represent the positions of those items in the sequence, starting with 1.

In the following example, the **for** clause includes a variable called `$cat` and an expression that constructs the sequence ("Persian", "Calico", "Siamese"). The clause also includes the positional variable `$i`, which is referenced in an attribute constructor to compute the value of the order attribute:

```
for $cat at $i in ("Persian", "Calico", "Siamese")
return <cat order = "{$i}"> { $cat } </cat>
```

When the **for** clause is evaluated, three tuples of variable bindings are created, each of which includes a binding for the positional variable:

```
($i = 1, $cat = "Persian")
($i = 2, $cat = "Calico")
($i = 3, $cat = "Siamese")
```

The **return** clause in the example executes once for each tuple of bindings. The expression results in the following output:

```
<cat order = "1">Persian</cat>
<cat order = "2">Calico</cat>
<cat order = "3">Siamese</cat>
```

Although each output element contains an order attribute, the actual order of the elements in the output stream is not guaranteed unless the FLWOR expression contains an **order by** clause such as `order by $i`. The positional variable represents the ordinal position of a value in the input sequence, not in the output sequence.

### Related reference

“**for** and **let** clauses in the same expression” on page 92

“**for** and **let** clauses compared” on page 93

“Variable scope in **for** and **let** clauses” on page 93

## let clauses

A **let** clause binds a variable to the entire result of an expression. A **let** clause does not perform any iteration.

The simplest type of **let** clause contains one variable and an associated expression. In the following example, the **let** clause includes a variable called `$j` and an expression that constructs the sequence (1, 2, 3).

```
let $j := (1, 2, 3)
return <output>{$j}</output>
```

When the **let** clause is evaluated, a single binding is created for the entire sequence that results from evaluating the expression:

```
$j = 1 2 3
```

The **return** clause in the example executes once. The expression results in the following output:

```
<output>1 2 3</output>
```

A **let** clause can contain multiple variables. However, unlike a **for** clause, a **let** clause binds each variable to the result of its associated expression, without iteration. In the following example, a **let** clause contains two variables, `$a` and `$b`, and expressions that construct the sequences 1 2 and 4 5:

```
let $a := (1, 2), $b := (4, 5)
return <output>{$a, $b}</output>
```

When the **let** clause is evaluated, one tuple of variable bindings is created:

```
($a = 1 2, $b = 4 5)
```

The **return** clause in the example executes once for the tuple. The expression results in the following output:

```
<output>1 2 4 5</output>
```

When the binding expression evaluates to an empty sequence, a **let** binding is created, which contains the empty sequence.

### Related reference

“**for** and **let** clauses in the same expression”

“**for** and **let** clauses compared” on page 93

“Variable scope in **for** and **let** clauses” on page 93

## for and let clauses in the same expression

When a FLWOR expression contains both **for** and **let** clauses, the variable bindings that are generated by **let** clauses are added to the variable bindings that are generated by the **for** clauses.

In the following example, the **for** clause includes a variable called `$a` and an expression that constructs the sequence (1, 2, 3). The **let** clause includes a variable called `$b` and an expression that constructs the sequence (4, 5, 6):

```
for $a in (1, 2, 3)
let $b := (4, 5, 6)
return <output>{$a, $b}</output>
```

The **for** and **let** clauses in this example result in three tuples of bindings. The number of tuples is determined by the **for** clause.

```
($a = 1, $b = 4 5 6)
($a = 2, $b = 4 5 6)
($a = 3, $b = 4 5 6)
```

The **return** clause in the example executes once for each tuple of bindings. The expression results in the following output:

```
<output>1 4 5 6</output>
<output>2 4 5 6</output>
<output>3 4 5 6</output>
```

### Related reference

“Variable scope in **for** and **let** clauses”

“**for** and **let** clauses compared”

## for and let clauses compared

Although **for** and **let** clauses both bind variables, the manner in which variables are bound is different.

The following table provides examples that compare the results that are returned by FLWOR expressions that contain similar **for** and **let** clauses.

Table 31. Comparison of **for** and **let** clauses in FLWOR expressions

Description of query	FLWOR expression	Result
Bind a single variable	for \$i in ("a", "b", "c") return <output>{\$i}</output>	<output>a</output> <output>b</output> <output>c</output>
	let \$i := ("a", "b", "c") return <output>{\$i}</output>	<output>a b c</output>
Bind multiple variables	for \$i in ("a", "b"), \$j in ("c", "d") return <output>{\$i, \$j}</output>	<output>a c</output> <output>b c</output> <output>a d</output> <output>b d</output>
	let \$i := ("a", "b"), \$j := ("c", "d") return <output>{\$i, \$j}</output>	<output>a b c d</output>

**Note:** Because the expressions in this table do not include **order by** clauses, the order of the output elements is non-deterministic.

### Related reference

“Variable scope in **for** and **let** clauses”

“**for** and **let** clauses in the same expression” on page 92

## Variable scope in for and let clauses

A variable that is bound in a **for** or **let** clause is in scope for all of the sub-expressions of the FLWOR expression that appear after the variable binding. This means that a **for** or **let** clause can reference variables that are bound in earlier clauses or in earlier bindings in the same clause.

In the following example, a FLWOR expression has the following clauses:

- A **let** clause that binds the variable \$orders.
- A **for** clause that references \$orders and binds the variable \$i.
- Another **let** clause that references both \$orders and \$i and binds the variable \$c.

The example finds all of the distinct item numbers in a set of orders, and returns the number of orders for each distinct item number.

```
let $orders := db2-fn:xmlcolumn("ORDERS.XMLORDER")
for $i in fn:distinct-values($orders/order/itemno)
let $c := fn:count($orders/order[itemno = $i])
return
<ordercount>
  <itemno> {$i} </itemno>
  <count> {$c} </count>
</ordercount>
```

**Important:** The **for** and **let** clauses of a FLWOR expression cannot bind the same variable name more than once.

**Related reference**

“Variable references” on page 54

“**for** and **let** clauses compared” on page 93

“**for** and **let** clauses in the same expression” on page 92

## where clauses

A **where** clause in a FLWOR expression filters the tuples of variable bindings that are generated by the **for** and **let** clauses.

The **where** clause specifies a condition that is applied to each tuple of variable bindings. If the condition is true (that is, if the expression results in an effective boolean value of true), then the tuple is retained, and its bindings are used when the **return** clause executes. Otherwise, the tuple is discarded.

In the following example, the **for** clause binds the variables `$x` and `$y` to sequences of numeric values:

```
for $x in (1.5, 2.6, 1.9), $y in (.5, 1.6, 1.7)
  where ((fn:floor($x) eq 1) and (fn:floor($y) eq 1))
  return <output>{$x, $y}</output>
```

When the **for** clause is evaluated, nine tuples of variable bindings are created:

```
($x = 1.5, $y = .5)
($x = 2.6, $y = .5)
($x = 1.9, $y = .5)
($x = 1.5, $y = 1.6)
($x = 2.6, $y = 1.6)
($x = 1.9, $y = 1.6)
($x = 1.5, $y = 1.7)
($x = 2.6, $y = 1.7)
($x = 1.9, $y = 1.7)
```

The **where** clause filters these tuples, and the following tuples are retained:

```
($x = 1.5, $y = 1.6)
($x = 1.9, $y = 1.6)
($x = 1.5, $y = 1.7)
($x = 1.9, $y = 1.7)
```

The **return** clause executes once for each remaining tuple, and the expression results in the following output:

```
<output>1.5 1.6</output>
<output>1.9 1.6</output>
<output>1.5 1.7</output>
<output>1.9 1.7</output>
```

Because the expression in this example does not include an **order by** clause, the order of the output elements is non-deterministic.

**Related concepts**

“Effective Boolean value” on page 51

**Related reference**

“Syntax of FLWOR expressions” on page 89

“Variable scope in **for** and **let** clauses” on page 93

## order by clauses

An **order by** clause in a FLWOR expression specifies the order in which values are processed by the **return** clause. If no **order by** clause is present, the results of a FLWOR expression are returned in a non-deterministic order.

An **order by** clause contains one or more ordering specifications. Ordering specifications are used to reorder the tuples of variable bindings that are retained after being filtered by the **where** clause. The resulting order determines the order in which the **return** clause is evaluated.

Each ordering specification consists of an expression, which is evaluated to produce an ordering key, and an order modifier, which specifies the sort order (ascending or descending) for the ordering keys. The relative order of two tuples is determined by comparing the values of their ordering keys, working from left to right.

In the following example, a FLWOR expression includes an **order by** clause that sorts products in descending order based on their price:

```
<price_list>{
  for $prod in db2-fn:xmlcolumn('PRODUCT.DESCRPTION')/product/description
  order by xs:decimal($prod/price) descending
  return
  <product>{$prod/name, $prod/price}</product>}
</price_list>
```

During processing of the **order by** clause, the expression in the ordering specification is evaluated for each tuple that is generated by the **for** clause. For the first tuple, the value that is returned by the expression `xs:decimal($prod/price)` is 9.99. The expression is then evaluated for the next tuple, and the expression returns the value 19.99. Because the ordering specification indicates that items are sorted in descending order, the product with the price 19.99 sorts before the product with the price 9.99. This sorting process continues until all tuples are reordered. The **return** clause then executes once for each tuple in the reordered tuple stream.

When run against the PRODUCT.DESCRPTION table of the SAMPLE database, the query in the example returns the following result:

```
<price_list>
  <product>
    <name>Snow Shovel, Super Deluxe 26"</name>
    <price>49.99</price>
  </product>
  <product>
    <name>Snow Shovel, Deluxe 24"</name>
    <price>19.99</price></product>
  <product>
    <name>Snow Shovel, Basic 22"</name>
    <price>9.99</price>
  </product>
  <product>
    <name>Ice Scraper, Windshield 4" Wide</name>
    <price>3.99</price>
  </product>
</price_list>
```

In this example, the expression in the ordering specification constructs an `xs:decimal` value from the value of the `price` element. This type conversion is necessary because the type annotation of the `price` element in the XML schema is `xs:string`. Without this conversion, the result would use string ordering rather than numeric ordering.

Explicit type conversion is also required when the dynamic type of the ordering key value is `xdt:untypedAtomic` because the rules for comparing ordering keys dictate that untyped atomic data is treated as a string.

**Tip:** You can use an **order by** clause in a FLWOR expression to specify value ordering in a query that would otherwise not require iteration. For example, the following path expression returns a list of customerinfo elements with a customer ID (Cid) that is greater than 1000:

```
db2-fn:xmlcolumn('CUSTOMER.INFO')/customerinfo[@Cid > "1000"]
```

To return these items in ascending order by the name of the customer, however, you would need to specify a FLWOR expression that includes an **order by** clause:

```
for $custinfo in db2-fn:xmlcolumn('CUSTOMER.INFO')/customerinfo
where ($custinfo/@Cid > "1000")
order by $custinfo/name ascending
return $custinfo
```

The ordering key does not need be part of the output. The following query produces a list of product names, in descending order by price, but does not include the price in the output:

```
for $prod in db2-fn:xmlcolumn('PRODUCT.DESCRPTION')/product
order by $prod/description/price descending
return $prod/name
```

## Rules for comparing ordering specifications

The process of evaluating and comparing ordering specifications is based on the following rules:

- The expression in the ordering specification is evaluated and atomization is applied to the result. The result of atomization must be either a single atomic value or an empty sequence; otherwise an error is returned. The result of evaluating an ordering specification is called an ordering key.
- If the type of an ordering key is `xdt:untypedAtomic`, then that key is cast to the type `xs:string`. Consistently treating untyped values as strings enables the sorting process to begin without complete knowledge of the types of all of the values to be sorted.
- If the values that are generated by an ordering specification are not all of the same type, these values (keys) are converted to a common type by subtype substitution or type promotion. Keys are compared by converting them to the least common type that supports the **gt** operator. For example, if an ordering specification generates a list of keys that includes both `xs:anyURI` values and `xs:string` values, the keys are compared by using the **gt** operator of the `xs:string` type. If the ordering keys that are generated by a given ordering specification do not have a common type that supports the **gt** operator, an error results.
- The values of the ordering keys are used to determine the order in which tuples of bound variables are passed to the return clause for execution. The ordering of tuples is determined by comparing their ordering keys, working from left to right, by using the following rules:
  - If the sort order is ascending, tuples with ordering keys that are greater than other tuples sort after those tuples.
  - If the sort order is descending, tuples with ordering keys that are greater than other tuples sort before those tuples.

The greater-than relationship for ordering keys is defined as follows:

- An empty sequence is greater than all other values.
- NaN is interpreted as greater than all other values except the empty sequence.
- A value is greater than another value if, when the value is compared to another value, the **gt** operator returns true.
- Neither of the special floating-point values positive zero or negative zero is greater than the other because `+0.0 gt -0.0` and `-0.0 gt +0.0` are both false.

**Note:** Tuples whose ordering key is empty appear at the end of the output stream if the **ascending** option, which is the default, is specified, or at the beginning of the output stream if the **descending** option is specified.

### Related concepts

“Atomization” on page 50

“Subtype substitution” on page 50

“Type promotion” on page 51

#### Related reference

“Syntax of FLWOR expressions” on page 89

“Order of results in XQuery expressions” on page 48

## return clauses

A **return** clause generates the result of the FLWOR expression.

The **return** clause is evaluated once for each tuple of variable bindings that is generated by the other clauses of the FLWOR expression. The results of these evaluations are concatenated to form the result of the FLWOR expression. The order in which tuples of bound variables are processed by the **return** clause is non-deterministic unless the FLWOR expression contains an **order by** clause.

**Tip:** In **return** clauses, use parentheses to enclose expressions that contain top-level comma operators. Because FLWOR expressions have a higher precedence than the comma operator, expressions that contain top-level comma operators could result in errors or unexpected results if parentheses are not used.

#### Related reference

“Syntax of FLWOR expressions” on page 89

“**order by** clauses” on page 95

## FLWOR examples

These examples show to how to use FLWOR expressions in complete queries to perform joins, grouping, and aggregation.

### FLWOR expression that joins XML data

The following query joins XML data from the PRODUCT and PURCHASEORDER tables in the SAMPLE database to list the names of products ordered in purchase orders placed in 2005.

Because the elements in both the product documents and the PurchaseOrder documents are in the same namespace, the query begins by declaring a default namespace so that the element names in the query do not need prefixes. The **for** clause iterates over the PURCHASEORDER.PORDER column, specifically for purchase orders with OrderDate attribute value that starts with “2005”. For each purchase order, the **let** clause assigns the partid values to the \$parts variable. The **return** clause then lists the names of the products that are included in the purchase order.

```
declare default element namespace 'http://posample.org';
for $po in db2-fn:xmlcolumn('PURCHASEORDER.PORDER')
  /PurchaseOrder[fn:starts-with(@OrderDate, "2005")]
let $parts := $po/item/partid
return
  <ProductList PoNum = "{$po/@PoNum}">
    { db2-fn:xmlcolumn('PRODUCT.DESCRPTION')
      /product[@pid = $parts]/description/name }
  </ProductList>
```

The result of the query is the following:

```
<ProductList xmlns="http://posample.org" PoNum="5001">
<name>
  Snow Shovel, Deluxe 24 inch
</name>
<name>
  Snow Shovel, Super Deluxe 26 inch
```

```

</name>
<name>
  Ice Scraper, Windshield 4 inch
</name>
</ProductList>
<ProductList xmlns="http://posample.org" PoNum="5003">
<name>
  Snow Shovel, Basic 22 inch
</name>
</ProductList>
<ProductList xmlns="http://posample.org" PoNum="5004">
<name>
  Snow Shovel, Basic 22 inch
</name>
<name>
  Snow Shovel, Super Deluxe 26 inch
</name>
</ProductList>

```

## FLWOR expression that groups elements

The following query groups customer names in the CUSTOMER table of the SAMPLE database by city. The **for** clause iterates over the customerinfo documents and binds each city element to the variable \$city. For each city, the **let** clause binds the variable \$cust-names to an unordered list of all the customer names in that city. The query returns city elements that each contain the name of a city and the nested name elements of all of the customers who live in that city.

```

declare default element namespace 'http://posample.org';
for $city in fn:distinct-values(db2-fn:xmlcolumn('CUSTOMER.INFO')
  /customerinfo/addr/city)
let $cust-names := db2-fn:xmlcolumn('CUSTOMER.INFO')
  /customerinfo/name[../addr/city = $city]
order by $city
return <city>{$city, $cust-names} </city>

```

The result of the query is the following:

```

<city xmlns="http://posample.org">Aurora
  <name>Robert Shoemaker</name>
</city>
<city xmlns="http://posample.org">Markham
  <name>Kathy Smith</name>
  <name>Jim Noodle</name>
</city>
<city xmlns="http://posample.org">Toronto
  <name>Kathy Smith</name>
  <name>Matt Foreman</name>
  <name>Larry Menard</name>
</city>

```

## FLWOR expression that aggregates data

The following query returns the total revenue generated by each purchase order in 2005 and creates an HTML report.

The query iterates over each PurchaseOrder element with an order date in 2005 and binds the element to the variable \$po in the **for** clause. The path expression \$po/item/ then moves the context position to each item element within a PurchaseOrder element. The nested expression (price \* quantity) determines the total revenue for that item. The fn:sum function adds the resulting sequence of total revenue for each item. The **let** clause binds the result of the fn:sum function to the variable \$revenue. The **order by** clause sorts the results by total revenue for each purchase order. Finally, the **return** clause creates a row in the report table for each purchase order.



```

declare default element namespace 'http://posample.org';
<html>
<body>
<h1>PO totals</h1>
<table>
<thead><tr><th>PO Number</th><th>Status</th>
<th>Revenue</th></tr></thead>
<tbody>{
  for $po in db2-fn:xmlcolumn('PURCHASEORDER.PORDER')/
    PurchaseOrder[fn:starts-with(@OrderDate, "2005")]
  let $revenue := sum($po/item/(price * quantity))
  order by $revenue descending
  return
  <tr>
    <td>{string($po/@PoNum)}</td>
    <td>{string($po/@Status)}</td>
    <td>{$revenue}</td>
  </tr>
}</tbody>
</table>
</body>
</html>

```

The result of this query is the following:

```

<html xmlns="http://posample.org">
<body>
  <h1>PO totals</h1>
  <table>
    <thead>
      <tr>
        <th>PO Number</th>
        <th>Status</th>
        <th>Revenue</th>
      </tr>
    </thead>
    <tbody>
      <tr>
        <td>5004</td>
        <td>Shipped</td>
        <td>139.94</td>
      </tr>
      <tr>
        <td>5001</td>
        <td>Shipped</td>
        <td>123.96</td>
      </tr>
      <tr>
        <td>5003</td>
        <td>UnShipped</td>
        <td>9.99</td>
      </tr>
    </tbody>
  </table>
</body>
</html>

```

When viewed in a browser, the query output would look similar to the following:

### PO totals

PO Number	Status	Revenue
5004	Shipped	139.94
5001	Shipped	123.96

PO Number	Status	Revenue
5003	Unshipped	9.99

### Related reference

“Syntax of FLWOR expressions” on page 89

## Conditional expressions

Conditional expressions use the keywords **if**, **then**, and **else** to evaluate one of two expressions based on whether the value of a test expression is true or false.

### Syntax

► `if (TestExpression) then Expression else Expression` ◄

**if** The keyword that directly precedes the test expression.

#### *TestExpression*

An XQuery expression that determines which part of the conditional expression to evaluate.

#### **then**

If the effective boolean value of *TestExpression* is true, then the expression that follows this keyword is evaluated. The expression is not evaluated or checked for errors if the effective boolean value of the test expression is false.

#### **else**

If the effective boolean value of *TestExpression* is false, then the expression that follows this keyword is evaluated. The expression is not evaluated or checked for errors if the effective boolean value of the test expression is true.

#### *Expression*

Any XQuery expression. If the expression includes a top-level comma operator, then the expression must be enclosed in parentheses.

## Example

In the following example, the query constructs a list of product elements that include an attribute named `basic`. The value of the `basic` attribute is specified conditionally based on whether the value of the price element is less than 10:

```
declare namespace ns1= "http://posample.org";
for $prod in db2-fn:xmlcolumn('PRODUCT.DESCRPTION')/ns1:product/ns1:description
return (
  if (xs:decimal($prod/ns1:price) < 10)
  then <product basic = "true">{fn:data($prod/ns1:name)}</product>
  else <product basic = "false">{fn:data($prod/ns1:name)}</product>)
```

The query returns the following result:

```
<product basic="true">Snow Shovel, Basic 22</product>
<product basic="false">Snow Shovel, Deluxe 24</product>
<product basic="false">Snow Shovel, Super Deluxe 26</product>
<product basic="true">Ice Scraper, Windshield 4" Wide</product>
```

In this example, the test expression constructs an `xs:decimal` value from the value of the price element. The `xs:decimal` function is used to force a decimal comparison.

### Related concepts

“Effective Boolean value” on page 51

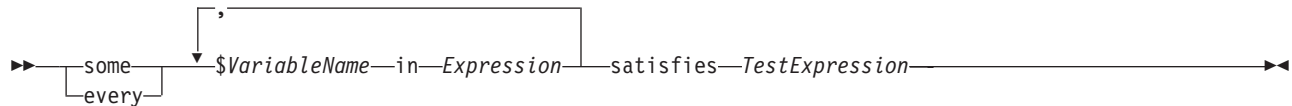
---

## Quantified expressions

Quantified expressions return true if some or every item in one or more sequences satisfies a specific condition. The value of a quantified expression is always true or false.

A quantified expression begins with a quantifier (**some** or **every**) that indicates whether the expression performs existential or universal quantification. The quantifier is followed by one or more clauses that bind variables to items that are returned by expressions. The bound variables are then referenced in a test expression to determine if some or all of the bound values satisfy a specific condition.

### Syntax



#### some

When this keyword is specified, the quantified expression returns true if the effective boolean value of *TestExpression* is true for *at least one* item that is returned by *Expression*. Otherwise, the quantified expression returns false.

#### every

When this keyword is specified, the quantified expression returns true if the effective boolean value of *TestExpression* is true for *every* item that is returned by *Expression*. Otherwise, the quantified expression returns false.

#### *VariableName*

The name of the variable to bind to each item in the result of *Expression*. Variables that are bound in a quantified expression are in scope for all of the sub-expressions that appear after the variable binding in the quantified expression.

#### *Expression*

Any XQuery expression. If the expression includes a top-level comma operator, then the expression must be enclosed in parentheses.

#### satisfies

The keyword that directly precedes the test expression

#### *TestExpression*

An XQuery expression that specifies the condition that must be met by some or every item in the sequences returned by *Expression*.

**Note:** When errors occur, the result of a quantified comparison can be either a boolean value or an error.

## Examples

- The quantified expression in the following example returns true if every customer in the CUSTOMER.INFO column of the SAMPLE database has an address in Canada:

```
every $cust in db2-fn:xmlcolumn('CUSTOMER.INFO')/customerinfo
satisfies $cust/addr/@country = "Canada"
```

- In the following examples, each quantified expression evaluates its test expression for every combination of values that are bound to the variables a and b (there are nine combinations in all).

The result of the following expression is true:

```
some $a in (3, 5, 9), $b in (1, 3, 5)
satisfies $a * $b = 27
```

The result of the following expression is false:

```
every $a in (3, 5, 9), $b in (1, 3, 5)
satisfies $a * $b = 27
```

- The following example demonstrates that the result of a quantified expression is not deterministic in the presence of errors. The expression can either return true or an error because the test expression returns true for one variable binding and returns an error for another:

```
some $a in (3, 5, "six") satisfies $a * 3 = 9
```

Likewise, the following expression can return false or an error:

```
every $a in (3, 5, "six") satisfies $a * 3 = 9
```

#### Related concepts

“Effective Boolean value” on page 51

#### Related reference

“Variable references” on page 54

## Cast expressions

A cast expression creates a new value of a specific type based on an existing value.

A cast expression takes two operands: an input expression and a target type. When the cast expression is evaluated, atomization is used to convert the result of the input expression into an atomic value or an empty sequence. If atomization results in a sequence of more than one atomic value, an error is returned. If no errors are returned, the cast expression attempts to create a new value of the target type that is based on the input value. Some combinations of input and target types are not supported for casting. For information about which types can be cast to which other types, see “Type casting” on page 23.

An empty sequence is a valid input value only when the target type is followed by a question mark (?).

### Syntax

►► *Expression* — cast as — *TargetType* [?] ◀◀

#### *Expression*

Any XQuery expression that returns a single atomic value or an empty sequence. An empty sequence is allowed when *TargetType* is followed by a question mark (?).

#### *TargetType*

The type to which the value of *Expression* is cast. *TargetType* must be an atomic type that is in the predefined atomic XML schema types. The data types `xs:NOTATION`, `xdt:anyAtomicType`, and `xs:anySimpleType` are not valid types for *TargetType*.

? Indicates that the result of *Expression* can be an empty sequence.

### Example

In the following example, a cast expression is used to cast the value of the price element, which has the type `xs:string`, to the type `xs:decimal`:

```
for $price in db2-fn:xmlcolumn('PRODUCT.DESCRPTION')/product/description/price
return $price cast as xs:decimal
```

When run against the `PRODUCT.DESCRPTION` table of the `SAMPLE` database, the query in the example returns the following result:

```
9.99
19.99
49.99
3.99
```

#### Related concepts

Chapter 2, “Type system,” on page 17

“Atomization” on page 50

**Related reference**

“Type casting” on page 23



---

## Chapter 5. Built-in functions

DB2 XQuery provides a library of built-in functions for working with XML data. These built-in functions include XQuery-defined functions and DB2-defined functions.

### XQuery-defined functions

XQuery-defined functions are in the namespace that is bound to the prefix `fn`. This namespace is the default function namespace, which means that you can invoke XQuery-defined functions without specifying a namespace prefix. If you override this default function namespace with a default function namespace declaration in the query prolog, you must use the prefix `fn` to invoke XQuery-defined functions.

### DB2-defined functions

The two DB2-defined functions are `db2-fn:xmlcolumn` and `db2-fn:sqlquery`, which you use to access XML values from a DB2 database. The prefix `db2-fn` is not the default function namespace, so you must use the namespace prefix when invoking these functions unless you override the default namespace with a default function namespace declaration in the query prolog.

#### Related concepts

“XML namespaces and QNames” on page 12

#### Related reference

“Default function namespace declaration” on page 42

---

## Functions by category

The following categories of functions are available: string, boolean, number, date, sequence, QName, node, and others.

### String functions

Function	Description
“codepoints-to-string function” on page 112	The <code>fn:codepoints-to-string</code> function returns the string equivalent of a sequence of Unicode code points.
“compare function” on page 113	The <code>fn:compare</code> function compares two strings.
“concat function” on page 114	The <code>fn:concat</code> function returns a string that is the concatenation of two or more atomic values.
“contains function” on page 114	The <code>fn:contains</code> function determines whether a string contains a given substring.
“ends-with function” on page 121	The <code>fn:ends-with</code> function determines whether a string ends with a given substring.
“lower-case function” on page 128	The <code>fn:lower-case</code> function converts a string to lowercase.
“matches function” on page 129	The <code>fn:matches</code> function determines whether a string matches a given pattern.
“normalize-space function” on page 135	The <code>fn:normalize-space</code> function strips leading and trailing whitespace characters from a string and replaces each internal sequence of whitespace characters with a single blank character.
“normalize-unicode function” on page 136	The <code>fn:normalize-unicode</code> function performs Unicode normalization on a string.
“replace function” on page 140	The <code>fn:replace</code> function compares each set of characters within a string to a given pattern, and then it replaces the characters that match the pattern with another set of characters.

Function	Description
“starts-with function” on page 147	The fn:starts-with function determines whether a string begins with a given substring.
“string function” on page 148	The fn:string function returns the string representation of a value.
“string-join function” on page 148	The fn:string-join function returns a string that is generated by concatenating items separated by a separator character.
“string-length function” on page 149	The fn:string-length function returns the length of a string.
“string-to-codepoints function” on page 149	The fn:string-to-codepoints function returns a sequence of Unicode code points that correspond to a string value.
“substring function” on page 151	The fn:substring function returns a substring of a string.
“substring-after function” on page 151	The fn:substring-after function returns a substring that occurs in a string after the end of the first occurrence of a given search string.
“substring-before function” on page 152	The fn:substring-before function returns a substring that occurs in a string before the first occurrence of a given search string.
“tokenize function” on page 154	The fn:tokenize function breaks a string into a sequence of substrings.
“translate function” on page 155	The fn:translate function replaces selected characters in a string with replacement characters.
“upper-case function” on page 157	The fn:upper-case function converts a string to uppercase.

## Boolean functions

Function	Description
“boolean function” on page 111	The fn:boolean function returns the effective Boolean value of a sequence.
“false function” on page 123	The fn:false function returns the xs:boolean value false.
“not function” on page 137	The fn:not function returns false if the effective boolean value of a sequence is true, and true if the effective boolean value of a sequence is false.
“true function” on page 156	The fn:true function returns the xs:boolean value true.
“zero-or-one function” on page 159	The fn:zero-or-one function returns its argument if the argument contains one item or is the empty sequence.

## Number functions

Function	Description
“abs function” on page 109	The fn:abs function returns the absolute value of a numeric value.
“avg function” on page 110	The fn:avg function returns the average of the values in a sequence.
“ceiling function” on page 111	The fn:ceiling function returns the smallest integer that is greater than or equal to a given numeric value.
“floor function” on page 123	The fn:floor function returns the largest integer that is less than or equal to a given numeric value.



Function	Description
“max function” on page 130	The fn:max function returns the maximum of the values in a sequence.
“min function” on page 131	The fn:min function returns the minimum of the values in a sequence.
“number function” on page 137	The fn:number function converts a value to the xs:double data type.
“round function” on page 144	The fn:round function returns the integer that is closest to a given numeric value.
“round-half-to-even function” on page 145	The fn:round-half-to-even function returns the numeric value with a specified precision that is closest to a given numeric value.
“sum function” on page 153	The fn:sum function returns the sum of the values in a sequence.

## Date functions

Function	Description
“current-date function” on page 115	The fn:current-date function returns the current date in the implicit timezone of UTC.
“current-dateTime function” on page 116	The fn:current-dateTime function returns the current date and time in the implicit timezone of UTC.
“current-time function” on page 116	The fn:current-time function returns the current time in the implicit timezone of UTC.
“dateTime function” on page 117	The fn:dateTime function constructs an xs:dateTime value from an xs:date value and an xs:time value.
“implicit-timezone function” on page 124	The fn:implicit-timezone function returns the implicit timezone value of PT0S, which is of type xs:dayTimeDuration. The value PT0S indicates that UTC is the implicit timezone.

## Sequence functions

Function	Description
“count function” on page 115	The fn:count function returns the number of values in a sequence.
“data function” on page 117	The fn:data function returns the input sequence after replacing any nodes in the input sequence by their typed values.
“deep-equal function” on page 118	The fn:deep-equal function compares two sequences to determine whether they meet the requirements for deep equality.
“distinct-values function” on page 120	The fn:distinct-values function returns the distinct values in a sequence.
“empty function” on page 121	The fn:empty function indicates whether the argument is an empty sequence.
“exactly-one function” on page 122	The fn:exactly-one function returns its argument if the argument contains exactly one item.
“exists function” on page 122	The fn:exists function indicates whether a sequence is not the empty sequence.

Function	Description
“last function” on page 126	The fn:last function returns the number of values in the sequence that is currently being processed.
“index-of function” on page 125	The fn:index-of function returns the positions where an item appears in a sequence.
“insert-before function” on page 126	The fn:insert-before function inserts a sequence before a given position in another sequence.
“one-or-more function” on page 138	The fn:one-or-more function returns its argument if the argument contains one or more items.
“position function” on page 138	The fn:position function returns the position of the context item in the sequence that is currently being processed.
“remove function” on page 140	The fn:remove function removes an item from a sequence.
“reverse function” on page 143	The fn:reverse function reverses the order of the items in a sequence.
“subsequence function” on page 150	The fn:subsequence function returns a subsequence of a sequence.
“unordered function” on page 157	The fn:unordered function returns the items in a sequence in non-deterministic order.

## QName functions

Function	Description
“in-scope-prefixes function” on page 124	The fn:in-scope-prefixes function returns a list of prefixes for all in-scope namespaces of an element.
“local-name-from-QName function” on page 128	The fn:local-name-from-QName function returns the local part of an xs:QName value.
“namespace-uri-for-prefix function” on page 134	The fn:namespace-uri-for-prefix function returns the namespace URI that is associated with a prefix in the in-scope namespaces for an element.
“namespace-uri-from-QName function” on page 134	The fn:namespace-uri-from-QName function returns the namespace URI part of an xs:QName value.
“QName function” on page 139	The fn:QName function builds an expanded name from a namespace URI and a string that contains a lexical QName (with an optional prefix.) .
“resolve-QName function” on page 142	The fn:resolve-QName function converts a string containing a lexical QName into an expanded QName by using the in-scope namespaces of an element to resolve the namespace prefix to a namespace URI.

## Node functions

Function	Description
“local-name function” on page 127	The fn:local-name function returns the local name property of a node.
“name function” on page 132	The fn:name function returns the prefix and local name parts of a node name.
“namespace-uri function” on page 133	The fn:namespace-uri function returns the namespace URI of the qualified name for a node.

Function	Description
“node-name function” on page 135	The fn:node-name function returns the expanded QName of a node.
“root function” on page 143	The fn:root function returns the root node of a tree to which a node belongs.

## Other functions

Function	Description
“default-collation function” on page 119	The fn:default-collation function returns a URI that represents the default collation that is defined for the database.
“sqlquery function” on page 146	The db2-fn:sqlquery function retrieves a sequence that is the result of an SQL fullselect in the currently connected DB2 database.
“xmlcolumn function” on page 158	The db2-fn:xmlcolumn function retrieves a sequence from a column in the currently connected DB2 database.

### Related reference

“Default function namespace declaration” on page 42

---

## abs function

The fn:abs function returns the absolute value of a numeric value.

### Syntax

►►—fn:abs(*numeric-value*)—◀◀

#### *numeric-value*

An atomic value or an empty sequence.

If *numeric-value* is an atomic value, it has one of the following types:

- xs:float
- xs:double
- xs:decimal
- xs:integer
- A type that is derived from any of the previously listed types
- xdt:untypedAtomic

If *numeric-value* has the xdt:untypedAtomic data type, it is converted to an xs:double value.

### Returned value

If *numeric-value* is not the empty sequence, the returned value is the absolute value of *numeric-value*.

If *numeric-value* is the empty sequence, fn:abs returns the empty sequence.

The data type of the returned value depends on the data type of *numeric-value*:

- If *numeric-value* is xs:float, xs:double, xs:decimal, or xs:integer, the value that is returned has the same type as *numeric-value*.
- If *numeric-value* has a data type that is derived from xs:float, xs:double, xs:decimal, or xs:integer, the value that is returned has the direct parent data type of *numeric-value*.

- If *numeric-value* has the xdt:untypedAtomic data type, the value that is returned has the xs:double data type.

## Example

The following function returns the absolute value of -10.5.

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

The returned value is 10.5.

### Related concepts

Chapter 2, "Type system," on page 17

---

## avg function

The fn:avg function returns the average of the values in a sequence.

### Syntax

►►—fn:avg(*sequence-expression*)—►►

#### *sequence-expression*

A sequence that contains items of any of the following atomic types, or an empty sequence:

- xs:float
- xs:double
- xs:decimal
- xs:integer
- xdt:untypedAtomic
- xdt:dayTimeDuration
- xdt:yearMonthDuration
- A type that is derived from any of the previously listed types

Input items of type xdt:untypedAtomic are cast to xs:double. After this casting, all of the items in the input sequence must be convertible to a common type by promotion or subtype substitution. The average is computed in this common type. For example, if the input sequence contains items of type money (derived from xs:decimal) and stockprice (derived from xs:float), the average is computed in the type xs:float.

### Returned value

If *sequence-expression* is not the empty sequence, the returned value is the average of the values in *sequence-expression*. The data type of the returned value is the same as the data type of the items in *sequence-expression*, or the data type to which the items in *sequence-expression* are promoted.

If *sequence-expression* is the empty sequence, the empty sequence is returned.

### Example

The following function returns the average of the sequence (5, 1.0E2, 40.5):

```
fn:avg((5, 1.0E2, 40.5))
```

The values are promoted to the xs:double data type. The function returns the xs:double value 4.85E1, which is serialized as "48.5".

### Related concepts

"Sequences and items" on page 4

"Type promotion" on page 51

---

## boolean function

The `fn:boolean` function returns the effective Boolean value of a sequence.

### Syntax

►► `fn:boolean(sequence-expression)` ◀◀

*sequence-expression*

Any sequence that contains items of any type, or the empty sequence.

### Returned value

The returned effective Boolean value depends on the value of *sequence-expression*:

Table 32. EBVs returned for specific types of values in XQuery

Description of value	EBV returned
An empty sequence	false
A sequence whose first item is a node	true
A single value of type <code>xs:boolean</code> (or derived from <code>xs:boolean</code> )	false - if the <code>xs:boolean</code> value is false true - if the <code>xs:boolean</code> value is true
A single value of type <code>xs:string</code> or <code>xdt:untypedAtomic</code> (or derived from one of these types)	false - if the length of the value is zero true - if the length of the value is greater than zero
A single value of any numeric type (or derived from a numeric type)	false - if the value is NaN or is numerically equal to zero true - if the value is not numerically equal to zero
All other values	error

**Note:** The effective boolean value of a sequence that contains at least one node and at least one atomic value is nondeterministic in a query where the order is unpredictable.

### Examples

**Example with an argument that is a single numeric value:** The following function returns the effective Boolean value of 0:

```
fn:boolean(0)
```

The returned value is false.

**Example with an argument that is a multiple-item sequence:** The following function returns the effective Boolean value of (`<a/>`, 0, `<b/>`):

```
fn:boolean((<a/>, 0, <b/>))
```

The returned value is true.

#### Related concepts

“Effective Boolean value” on page 51

“Sequences and items” on page 4

---

## ceiling function

The `fn:ceiling` function returns the smallest integer that is greater than or equal to a given numeric value.

## Syntax

►—fn:ceiling(*numeric-value*)—◄

### *numeric-value*

An atomic value or an empty sequence.

If *numeric-value* is an atomic value, it has one of the following types:

- xs:float
- xs:double
- xs:decimal
- xs:integer
- xdt:untypedAtomic
- A type that is derived from any of the previously listed types

If *numeric-value* has the xdt:untypedAtomic data type, it is converted to an xs:double value.

## Returned value

If *numeric-value* is not the empty sequence, the returned value is the smallest integer that is greater than or equal to *numeric-value*. The data type of the returned value depends on the data type of *numeric-value*:

- If *numeric-value* is xs:float, xs:double, xs:decimal, or xs:integer, the value that is returned has the same type as *numeric-value*.
- If *numeric-value* has a data type that is derived from xs:float, xs:double, xs:decimal, or xs:integer, the value that is returned has the direct parent data type of *numeric-value*.

If *numeric-value* is the empty sequence, the returned value is the empty sequence.

## Examples

**Example with a positive argument:** The following function returns the ceiling value of 0.5:

```
fn:ceiling(0.5)
```

The returned value is 1.

**Example with a negative argument:** The following function returns the ceiling value of (-1.2):

```
fn:ceiling(-1.2)
```

The returned value is -1.

### Related concepts

Chapter 2, “Type system,” on page 17

---

## codepoints-to-string function

The fn:codepoints-to-string function returns the string equivalent of a sequence of Unicode code points.

## Syntax

►—fn:codepoints-to-string(*codepoint-sequence*)—◄

### *codepoint-sequence*

A sequence of integers that correspond to Unicode code points, or the empty sequence.

## Returned value

If *codepoint-sequence* is not the empty sequence, the returned value is a string that is the concatenation of the character equivalents of the items in *codepoint-sequence*. If any item in *codepoint-sequence* is not a valid Unicode code point, an error is returned.

If *codepoint-sequence* is the empty sequence, the returned value is a string of length 0.

## Example

The following function returns the character equivalent of the sequence of UTF-8 code points (88,81,117,101,114,121).

```
fn:codepoints-to-string((88,81,117,101,114,121))
```

The returned value is 'XQuery'.

### Related reference

“string-to-codepoints function” on page 149

---

## compare function

The fn:compare function compares two strings.

## Syntax

```
fn:compare(string-1,string-2)
```

*string-1* , *string-2*

The xs:string values that are to be compared.

## Returned value

If *string-1* and *string-2* are not the empty sequence, one of the following values is returned:

- 1 If *string-1* is less than *string-2*.
- 0 If *string-1* is equal to *string-2*.
- 1 If *string-1* is greater than *string-2*.

*string-1* and *string-2* are equal if they have the same length, including a length of zero, and all corresponding characters are equal according to the default collation.

If *string-1* and *string-2* are not equal, their relationship (that is, which has the greater value) is determined by the comparison of the first pair of unequal characters from the left end of the strings. This comparison is made according to the default collation.

If *string-1* is longer than *string-2*, and all characters of *string-2* are equal to the leading characters of *string-1*, *string-1* is greater than *string-2*.

If *string-1* or *string-2* is the empty sequence, the empty sequence is returned.

## Example

The following function compares 'ABC' to 'ABD' using the default collation.

```
fn:compare('ABC', 'ABD')
```

'ABC' is less than 'ABD'. The returned value is -1.

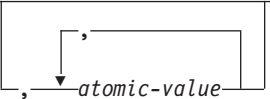
---

## concat function

The fn:concat function returns a string that is the concatenation of two or more atomic values.

### Syntax

►►—fn:concat(*atomic-value*,*atomic-value*—)◀◀



#### *atomic-value*

An atomic value or the empty sequence. If an argument is the empty sequence, the argument is treated as the zero-length string. If *atomic-value* is not an xs:string value, it is cast to xs:string before the values are concatenated.

### Returned value

If all *atomic-value* arguments are the empty sequence, the returned value is a string of length 0. Otherwise, the returned value is the concatenation of the xs:string values that result from casting the *atomic-value* arguments to strings.

### Example

The following function concatenates the strings 'ABC', 'ABD', the empty sequence, and 'ABE':

```
fn:concat('ABC', 'ABD', (), 'ABE')
```

The returned value is 'ABCABDABE'.

#### **Related concepts**

“Atomic values” on page 4

#### **Related reference**

“tokenize function” on page 154

---

## contains function

The fn:contains function determines whether a string contains a given substring.

### Syntax

►►—fn:contains(*string*,*substring*)◀◀

*string* The string to search for *substring*.

*string* has the xs:string data type, or is the empty sequence. If *string* is the empty sequence, *string* is set to a string of length 0.

#### *substring*

The substring to search for in *string*.

*substring* has the xs:string data type, or is the empty sequence.

### Returned value

The returned value is the xs:boolean value true if either of the following conditions are satisfied:

- *substring* occurs anywhere within *string*.
- *substring* is an empty sequence or a string of length zero.



Otherwise, the returned value is false.

## Example

The following function determines whether the string 'Test literal' contains the string 'lite'.

```
fn:contains('Test literal','lite')
```

The returned value is true.

### Related reference

“ends-with function” on page 121

“starts-with function” on page 147

---

## count function

The fn:count function returns the number of values in a sequence.

### Syntax

```
►► fn:count(sequence-expression) ◀◀
```

*sequence-expression*

A sequence that contains items of any type, or an empty sequence.

### Returned value

If *sequence-expression* is not the empty sequence, the number of values in *sequence-expression* is returned. If *sequence-expression* is the empty sequence, 0 is returned.

## Example

The following function returns the number of items in the sequence (5, 1.0E2, 40.5):

```
fn:count((5, 1.0E2, 40.5))
```

The returned value is 3.

### Related concepts

“Sequences and items” on page 4

---

## current-date function

The fn:current-date function returns the current date in the implicit timezone of UTC.

### Syntax

```
►► fn:current-date() ◀◀
```

### Returned value

The returned value is an xs:date value that is the current date.

## Example

The following function returns the current date.

```
fn:current-date()
```

If this function were invoked on December 2, 2005, the returned value would be 2005-12-02Z.

**Related reference**

“current-dateTime function”

“current-time function”

“dateTime function” on page 117

“implicit-timezone function” on page 124

---

## current-dateTime function

The fn:current-dateTime function returns the current date and time in the implicit timezone of UTC.

### Syntax

►—fn:current-dateTime()—◄

### Returned value

The returned value is an xs:dateTime value that is the current date and time.

### Example

The following function returns the current date and time.

```
fn:current-dateTime()
```

If this function were invoked on December 2, 2005 at 6:25 in Toronto (timezone -PT5H), the returned value might be 2005-12-02T011:25:30.864001Z.

**Related reference**

“current-date function” on page 115

“current-time function”

“dateTime function” on page 117

“implicit-timezone function” on page 124

---

## current-time function

The fn:current-time function returns the current time in the implicit timezone of UTC.

### Syntax

►—fn:current-time()—◄

### Returned value

The returned value is an xs:time value that is the current time.

### Example

The following function returns the current time.

```
fn:current-time()
```

If this function were invoked at 6:31 Greenwich Mean Time, the returned value might be 06:31:35.519001Z.

**Related reference**

- “current-date function” on page 115
- “current-dateTime function” on page 116
- “dateTime function”
- “implicit-timezone function” on page 124

---

## data function

The `fn:data` function returns the input sequence after replacing any nodes in the input sequence by their typed values.

### Syntax

►► `fn:data(sequence-expression)` ◀◀

*sequence-expression*

Any sequence, including the empty sequence.

### Returned value

If *sequence-expression* is an empty sequence, the returned value is an empty sequence.

If *sequence-expression* is a single atomic value, the returned value is *sequence-expression*.

If *sequence-expression* is a single node, the returned value is the typed value of *sequence-expression*.

If *sequence-expression* is a sequence of more than one item, a sequence of atomic values is returned from the items in *sequence-expression*. Each atomic value in *sequence-expression* remains unchanged. Each node in *sequence-expression* is replaced by its typed value, which is a sequence of zero or more atomic values.

### Example

The following function returns a sequence that contains the atomic values that are in the sequence (`<x xsi:type="string">ABC</x>,<y xsi:type="decimal">1.23</y>`).

```
fn:data((<x xsi:type="string">ABC</x>,<y xsi:type="decimal">1.23</y>))
```

The returned value is ("ABC",1.23).

#### Related concepts

“Typed values and string values of nodes” on page 10

---

## dateTime function

The `fn:dateTime` function constructs an `xs:dateTime` value from an `xs:date` value and an `xs:time` value.

### Syntax

►► `fn:dateTime(date-value,time-value)` ◀◀

*date-value*

An `xs:date` value.

*time-value*

An `xs:time` value.

## Returned value

The returned value is an `xs:dateTime` value with a date component that is equal to *date-value* and a time component that is equal to *time-value*. The timezone of the result is computed as follows:

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

## Example

The following function returns an `xs:dateTime` value from an `xs:date` value and an `xs:time` value.

```
fn:dateTime((xs:date("2005-04-16")), (xs:time("12:30:59")))
```

The returned value is the `xs:dateTime` value 2005-04-16T12:30:59.

### Related reference

“current-date function” on page 115

“current-time function” on page 116

“implicit-timezone function” on page 124

---

## deep-equal function

The `fn:deep-equal` function compares two sequences to determine whether they meet the requirements for deep equality.

### Syntax

```
►►—fn:deep-equal(sequence-1,sequence-2)—◄◄
```

*sequence-1*, *sequence-2*

The sequences that are to be compared. The items in each sequence can be atomic values of any type, or nodes.

### Returned value

The returned value is the `xs:boolean` value `true` if *sequence-1* and *sequence-2* have deep equality. Otherwise the returned value is `false`.

If *sequence-1* and *sequence-2* are the empty sequence, they have deep equality.

If two sequences are not empty, the two sequences have deep equality if they satisfy both of the following conditions:

- The number of items in *sequence-1* is equal to the number of items in *sequence-2*.
- Each item in *sequence-1* (*item-1*) satisfies the conditions for deep equality to the corresponding item in *sequence-2* (*item-2*). *item-1* and *item-2* have deep equality if they satisfy either of the following conditions:
  - *item-1* and *item-2* are both atomic values and satisfy either of the following conditions:
    - The expression `item-1 eq item-2` returns `true`
    - Both *item-1* and *item-2* have the type `xs:float` or `xs:double` and the value `NaN`.
  - *item-1* and *item-2* are both nodes of the same kind and satisfy the conditions for deep equality in the following table.

Table 33. Deep equality for nodes in a sequence

Node kind of both item-1 and item-2	Conditions for deep equality
Document	The sequence of the text and element children of item-1 is deep-equal to the sequence of the text and element children of item-2.
Element	<p>All of the following conditions must be true:</p> <ul style="list-style-type: none"> <li>• <i>item-1</i> and <i>item-2</i> have the same name, which means that their namespace URIs match and their local names match. Namespace prefixes are ignored.</li> <li>• <i>item-1</i> and <i>item-2</i> have the same number of attributes, and every attribute of <i>item-1</i> is deep-equal to an attribute of <i>item-2</i>.</li> <li>• One of the following conditions is true: <ul style="list-style-type: none"> <li>– Both nodes are either unvalidated or validated with a type that permits mixed content (both text and child elements), and the sequence of the text and element children of <i>item-1</i> is deep-equal to the sequence of the text and element children of <i>item-2</i>.</li> <li>– Both nodes are validated with a simple type (such as <code>xs:decimal</code>) or a type that has simple content (such as a "temperature" type whose content is <code>xs:decimal</code>), and the typed value of <i>item-1</i> is deep-equal to the typed value of <i>item-2</i>.</li> <li>– Both nodes are validated with a type that permits no content (neither text nor child elements).</li> <li>– Both nodes are validated with a type that permits only child elements (no text), and each child element of <i>item-1</i> is deep-equal to the corresponding child element of <i>item-2</i>.</li> </ul> </li> </ul>
Attribute	<p>All of the following conditions must be true:</p> <ul style="list-style-type: none"> <li>• <i>item-1</i> and <i>item-2</i> have the same name, which means that their namespace URIs match and their local names match. Namespace prefixes are ignored.</li> <li>• The typed value of <i>item-1</i> is deep-equal to the typed value of <i>item-2</i>.</li> </ul>
Text	The content property values are equal when compared as strings with the eq operator.
Comment	The content property values are equal when compared as strings with the eq operator.
Processing instruction	<p>All of the following conditions must be true:</p> <ul style="list-style-type: none"> <li>• <i>item-1</i> and <i>item-2</i> have the same name.</li> <li>• The content property values are equal when compared as strings with the eq operator.</li> </ul>

## Example

The following function compares the sequences (1,'ABC') and (1,'ABCD') for deep equality. String comparisons use the default correlation.

```
fn:deep-equal((1,'ABC'), (1,'ABCD'))
```

The returned value is false.

### Related concepts

“Typed values and string values of nodes” on page 10

“Node kinds” on page 7

### Related reference

“General comparisons” on page 71

---

## default-collation function

The `fn:default-collation` function returns a URI that represents the default collation that is defined for the database.

## Syntax

►—fn:default-collation()—◄

## Returned value

The returned value is of the type `xs:anyURI` and specifies the collation of the database.

## Example

A DB2 database is created with the collation `UCA400_NO` specified. When querying this database, the following function returns `http://www.ibm.com/xmlns/prod/db2/sql/collations?name=UCA400_NO`:

```
fn:default-collation()
```

---

## distinct-values function

The `fn:distinct-values` function returns the distinct values in a sequence.

## Syntax

►—fn:distinct-values(*sequence-expression*)—◄

*sequence-expression*

A sequence of atomic values, or the empty sequence.

## Returned value

If *sequence-expression* is not the empty sequence, the returned value is a sequence that contains the distinct values in *sequence-expression*. Two values, *value1* and *value2*, are distinct if *value1* `eq` *value2* is false. If the `eq` operator is not defined for two values, those values are considered to be distinct.

Values of type `xdt:untypedAtomic` are converted to values of type `xs:string` before the values are compared.

For `xs:float` and `xs:double` values, if *sequence-expression* contains multiple NaN values, a single NaN value is returned.

For `xs:dateTime`, `xs:date`, or `xs:time` values, the values are adjusted for timezone differences before they are compared. If a value does not have a timezone, the implicit timezone (UTC) is used.

If *sequence-expression* is the empty sequence, the empty sequence is returned.

If two values in the input sequence are equal by the `eq` operator but have different types, either of the values, but not both, can appear in the result sequence. The result sequence might not preserve the order of the input sequence.

## Example

The following function returns the distinct values in a sequence, after atomizing the nodes in the sequence:

```
fn:distinct-values((1, 'a', 1.0, 'A', <greeting>Hello</greeting>))
```

The returned value may be `(1, 'a', 'A', 'Hello')` or `(1.0, 'A', 'a', 'Hello')`.

### Related reference

“Value comparisons” on page 69

---

## empty function

The `fn:empty` function indicates whether the argument is an empty sequence.

### Syntax

►► `fn:empty(item)` ◀◀

*item* An expression of any data type, or the empty sequence.

### Returned value

The returned value is true if *item* is the empty sequence. Otherwise, the returned value is false.

### Example

The following example uses the empty function to determine whether the sequence in variable `$seq` is the empty sequence.

```
let $seq := (5, 10)
return fn:empty($seq)
```

The returned value is false.

#### Related concepts

“Sequences and items” on page 4

---

## ends-with function

The `fn:ends-with` function determines whether a string ends with a given substring.

### Syntax

►► `fn:ends-with(string,substring)` ◀◀

*string* The string to search for *substring*.

*string* has the `xs:string` data type, or is an empty sequence. If *string* is an empty sequence, *string* is set to a string of length 0.

*substring*

The substring to search for at the end of *string*.

*substring* has the `xs:string` data type, or is an empty sequence.

### Returned value

The returned value is the `xs:boolean` value true if either of the following conditions is satisfied:

- *substring* occurs at the end of *string*.
- *substring* is an empty sequence or a string of length zero.

Otherwise, the returned value is false.

### Example

The following function determines whether the string 'Test literal' ends with the string 'literal'.

```
fn:ends-with('Test literal','literal')
```

The returned value is true.

**Related reference**

“contains function” on page 114

“starts-with function” on page 147

---

**exactly-one function**

The `fn:exactly-one` function returns its argument if the argument contains exactly one item.

**Syntax**

►—`fn:exactly-one(sequence-expression)`—◄

*sequence-expression*

Any sequence, including the empty sequence.

**Returned value**

If *sequence-expression* contains exactly one item, *sequence-expression* is returned. Otherwise, an error is returned.

**Example**

The following example uses the `exactly-one` function to determine whether the sequence in variable `$seq` contains exactly one item.

```
let $seq := 5
return fn:exactly-one($seq)
```

The value 5 is returned.

**Related concepts**

“Sequences and items” on page 4

**Related reference**

“one-or-more function” on page 138

“zero-or-one function” on page 159

---

**exists function**

The `fn:exists` function indicates whether a sequence is not the empty sequence.

**Syntax**

►—`fn:exists(sequence-expression)`—◄

*sequence-expression*

A sequence of any data type, or the empty sequence

**Returned value**

The returned value is true if *sequence-expression* is not the empty sequence. Otherwise, the returned value is false.

**Example**

The following example uses the `exists` function to determine whether the sequence in variable `$seq` is not the empty sequence.



```
let $seq := (5, 10)
return fn:exists($seq)
```

The value true is returned.

#### Related concepts

“Sequences and items” on page 4

---

## false function

The fn:false function returns the xs:boolean value false.

### Syntax

►► fn:false()

---

### Returned value

The returned value is the xs:boolean value false.

### Example

Use the false function to return the value false.

```
fn:false()
```

The value false is returned.

#### Related reference

“true function” on page 156

---

## floor function

The fn:floor function returns the largest integer that is less than or equal to a given numeric value.

### Syntax

►► fn:floor(*numeric-value*)

---

#### *numeric-value*

An atomic value or an empty sequence.

If *numeric-value* is an atomic value, it has one of the following types:

- xs:float
- xs:double
- xs:decimal
- xs:integer
- xdt:untypedAtomic
- A type that is derived from any of the previously listed types

If *numeric-value* has the xdt:untypedAtomic data type, it is converted to an xs:double value.

### Returned value

If *numeric-value* is not the empty sequence, the returned value is the largest integer that is less than *numeric-value*. The data type of the returned value depends on the data type of *numeric-value*:

- If *numeric-value* is xs:float, xs:double, xs:decimal, or xs:integer, the value that is returned has the same type as *numeric-value*.

- If *numeric-value* has a data type that is derived from xs:float, xs:double, xs:decimal, or xs:integer, the value that is returned has the direct parent data type of *numeric-value*.

If *numeric-value* is the empty sequence, the returned value is the empty sequence.

## Examples

**Example with a positive argument:** The following function returns the floor value of 0.5:

```
fn:floor(0.5)
```

The returned value is 0.

**Example with a negative argument:** The following function returns the floor value of (-1.2):

```
fn:floor(-1.2)
```

The returned value is -2.

---

## implicit-timezone function

The fn:implicit-timezone function returns the implicit timezone value of PT0S, which is of type xs:dayTimeDuration. The value PT0S indicates that UTC is the implicit timezone.

### Syntax

```
►—fn:implicit-timezone()—◄
```

### Returned value

The returned value is PT0S, which is UTC represented by the type xs:dayTimeDuration.

### Example

The following function returns xdt:dayTimeDuration("PT0S"):

```
fn:implicit-timezone()
```

#### Related reference

“current-date function” on page 115

“current-dateTime function” on page 116

“current-time function” on page 116

“dateTime function” on page 117

---

## in-scope-prefixes function

The fn:in-scope-prefixes function returns a list of prefixes for all in-scope namespaces of an element.

### Syntax

```
►—fn:in-scope-prefixes(element)—◄
```

*element*

The element for which the prefixes for in-scope namespaces are to be retrieved.

## Returned value

The returned value is a sequence of `xs:NCName` values, which are the prefixes for all in-scope namespaces for *element*. If a default namespace is in-scope for *element*, the sequence item for the default namespace prefix is a string of length 0. The namespace "xml" is always included in the in-scope namespaces of an element.

## Example

The following query returns a sequences of prefixes (as NCNames) for in-scope namespaces for the element `emp`.

```
declare namespace d="http://www.example.org";
let $department := document {
<comp:dept xmlns:comp="http://www.mycompany.com" id="A07">
  <comp:emp id="31201" />
</comp:dept> }
return fn:in-scope-prefixes($department/d:dept/d:emp)
```

The returned value is ("xml", "comp"), not necessarily in that order.

### Related reference

"In-scope namespaces of a constructed element" on page 82

---

## index-of function

The `fn:index-of` function returns the positions where an item appears in a sequence.

## Syntax

►► `fn:index-of(sequence-expression,search-value)` ◀◀

### *sequence-expression*

Any sequence of atomic types, or the empty sequence.

### *search-value*

The value to find in *sequence-expression*.

## Returned value

The returned value is a sequence of `xs:integer` values that represent the positions of items in *sequence-expression* that match *search-value* when compared by using the rules of the `eq` operator. Items that cannot be compared because the `eq` operator is not defined for their types are considered to not match *search-value*, and therefore the positions are not returned. The first item in a sequence has the position 1.

The function returns an empty sequence if *search-value* does not match any items in *sequence-expression*, or if *sequence-expression* is an empty sequence.

## Example

The following function returns the positions where 'ABC' appears in a sequence.

```
fn:index-of(('ABC','DEF','ABC','123'),'ABC')
```

The returned value is the sequence (1,3).

### Related concepts

"Sequences and items" on page 4

---

## insert-before function

The fn:insert-before function inserts a sequence before a given position in another sequence.

### Syntax

►—fn:insert-before(*source-sequence*,*insert-position*,*insert-sequence*)—►

#### *source-sequence*

The sequence into which a sequence is to be inserted.

*source-sequence* is a sequence of items of any data type, or is the empty sequence.

#### *insert-position*

The position in *source-sequence* before which a sequence is to be inserted. *insert-position* has the xs:integer data type. If *insert-position* ≤ 0, *insert-position* is set to 1. If *insert-position* is greater than the number of items in *source-sequence*, *insert-position* is set to one greater than the number of items in *source-sequence*.

#### *insert-sequence*

The sequence that is to be inserted into *source-sequence*.

*insert-sequence* is a sequence of items of any data type, or is the empty sequence.

### Returned value

If *source-sequence* is not the empty sequence:

- If *insert-sequence* is not the empty sequence, the returned value is a sequence with the following items, in the following order:
  - The items in *source-sequence* before item *insert-position*
  - The items in *insert-sequence*
  - The item in *source-sequence* at item *insert-position*
  - The items in *source-sequence* after item *insert-position*
- If *insert-sequence* is the empty sequence, the returned value is *source-sequence*.

If *source-sequence* is the empty sequence:

- If *insert-sequence* is not an empty sequence, the returned value is *insert-sequence*.
- If *insert-sequence* is an empty sequence, the returned value is the empty sequence.

### Example

The following function returns the sequence that results from inserting the sequence (4,5,6) before position 4 in sequence (1,2,3,7):

```
fn:insert-before((1,2,3,7),4,(4,5,6))
```

The returned value is (1,2,3,4,5,6,7).

#### Related concepts

“Sequences and items” on page 4

---

## last function

The fn:last function returns the number of values in the sequence that is currently being processed.

### Syntax

►► fn:last() ◀◀

## Returned value

If the sequence that is currently being processed is not the empty sequence, the returned value is the number of values in the sequence. If the sequence that is currently being processed is the empty sequence, the returned value is the empty sequence.

## Example

This function returns the number of items in the current sequence, which is (5, 1.0E2, 40.5):

```
fn:last()
```

The returned value is 3.

### Related concepts

“Sequences and items” on page 4

### Related reference

“Dynamic context and focus” on page 47

---

## local-name function

The fn:local-name function returns the local name property of a node.

## Syntax

►► fn:local-name(node) ◀◀

*node* The node for which the local name is to be retrieved. If *node* is not specified, fn:local-name is evaluated for the current context node.

## Returned value

The returned value depends on whether *node* is specified, and the value of *node*:

- If *node* is not specified, the local name of the context node is returned.
- If *node* meets any of the following conditions, a string of length 0 is returned:
  - *node* is the empty sequence.
  - *node* is not an element node, an attribute node, or a processing-instruction node.
- If *node* meets any of the following conditions, an error is returned:
  - *node* is undefined.
  - *node* is not a node.
- Otherwise, an xs:string value is returned that contains the local name part of the expanded name for *node*.

## Examples

The following function returns the local name for node emp.

```
declare namespace a="http://posample.org";  
fn:local-name(<a:b/>)
```

The returned value is b.

### Related concepts

---

## local-name-from-QName function

The `fn:local-name-from-QName` function returns the local part of an `xs:QName` value.

### Syntax

►► `fn:local-name-from-QName(qualified-name)` ◀◀

#### *qualified-name*

The qualified name from which the local part is to be retrieved.

*qualified-name* has the `xs:QName` data type, or is the empty sequence.

### Returned value

If *qualified-name* is not the empty sequence, the value that is returned is an `xs:NCName` value that is the local part of *qualified-name*. If *qualified-name* is the empty sequence, the empty sequence is returned.

### Example

The following function returns the local part of a qualified name.

```
fn:local-name-from-QName(fn:QName("http://www.mycompany.com/", "ns:employee"))
```

The returned value is "employee".

#### Related concepts

“Qualified names (QNames)” on page 12

---

## lower-case function

The `fn:lower-case` function converts a string to lowercase.

### Syntax

►► `fn:lower-case(source-string)` ◀◀

#### *source-string*

The string that is to be converted to lowercase.

*source-string* has the `xs:string` data type, or is the empty sequence.

### Returned value

If *source-string* is not the empty sequence, the returned value is *source-string*, with each character converted to its lowercase correspondent as defined in the Unicode standard. Every character that does not have a lowercase correspondent is included in the returned value in its original form.

If *source-string* is the empty sequence, the returned value is a string of length 0.

### Example

The following function converts the string "Wireless Router TB2561" to lowercase:

```
fn:lower-case("Wireless Router TB2561")
```

Returns: "wireless router tb2561"

---

## matches function

The `fn:matches` function determines whether a string matches a given pattern.

### Syntax

► `fn:matches(source-string,pattern [ ,flags ] )` ◀

#### *source-string*

A string that is compared to a pattern.

*source-string* is an `xs:string` value or the empty sequence.

#### *pattern*

A regular expression that is compared to *source-string*. A regular expression is a set of characters, wildcards, and operators that define a string or group of strings in a search pattern.

*pattern* is an `xs:string` value.

*flags* An `xs:string` value that can contain any of the following values that control matching of *pattern* to *source-string*:

- s** Indicates that the dot (.) matches any character.  
If the `s` flag is not specified, the dot (.) matches any character except the new line character (X'0A').
- m** Indicates that the caret (^) matches the start of a line (the position after a new line character), and the dollar sign (\$) matches the end of a line (the position before a new line character).  
If the `m` flag is not specified, the caret (^) matches the start of a string, and the dollar sign (\$) matches the end of the string.
- i** Indicates that matching is case-insensitive.  
If the `i` flag is not specified, case-sensitive matching is done.
- x** Indicates that whitespace characters within *pattern* are ignored.  
If the `x` flag is not specified, whitespace characters are used for matching.

### Returned value

If *source-string* is not the empty sequence, the returned value is `true` if *source-string* matches *pattern*. The returned value is `false` if *source-string* does not match *pattern*.

If *pattern* does not contain the string- or line-starting character caret (^), or the string- or line-ending character dollar sign (\$), *source-string* matches *pattern* if any substring of *source-string* matches *pattern*. If *pattern* contains the string- or line-starting character caret (^), *source-string* matches *pattern* only if *source-string* matches *pattern* from the beginning of *source-string* or the beginning of a line in *source-string*. If *pattern* contains the string- or line-ending character dollar sign (\$), *source-string* matches *pattern* only if *source-string* matches *pattern* at the end of *source-string* or at the end of a line of *source-string*. The `m` flag determines whether the match occurs from the beginning of the string or the beginning of a line.

If *source-string* is the empty sequence, the returned value is `false`.

## Examples

**Example of matching a pattern to any substring within a string:** The following function determines whether the characters "ac" or "bd" appear anywhere within the string "abbcacadbdc".

```
fn:matches("abbcacadbdc", "(ac)|(bd)")
```

The returned value is true.

**Example of matching a pattern to an entire string:** The following function determines whether the characters "ac" or "bd" match the string "bd".

```
fn:matches("bd", "^(ac)|(bd)$")
```

The returned value is true.

### Related reference

Regular expressions

A regular expression is a sequence of characters that act as a pattern for matching and manipulating strings. Regular expressions are used in the following XQuery functions: `fn:matches`, `fn:replace`, and `fn:tokenize`. DB2 XQuery regular expression support is based on the XML schema regular expression support as defined in the W3C Recommendation *XML Schema Part 2: Datatypes Second Edition* with extensions as defined by W3C Candidate Recommendation *XQuery 1.0 and XPath 2.0 Functions and Operators*.

---

## max function

The `fn:max` function returns the maximum of the values in a sequence.

### Syntax

```
►►—fn:max(sequence-expression)—►►
```

#### *sequence-expression*

A sequence that contains items of any of the following atomic types, or an empty sequence:

- `xs:float`
- `xs:double`
- `xs:decimal`
- `xs:integer`
- `xs:string`
- `xs:date`
- `xs:time`
- `xs:dateTime`
- `xd:untypedAtomic`
- `xd:dayTimeDuration`
- `xd:yearMonthDuration`
- A type that is derived from any of the previously listed types

Input items of type `xd:untypedAtomic` are cast to `xs:double`. After this casting, all the items in the input sequence must be convertible by promotion or subtype substitution to a common type that supports the **ge** operator. The maximum value is computed in this common type. For example, if the input sequence contains items of type `money` (derived from `xs:decimal`) and `stockprice` (derived from `xs:float`), the maximum is computed in the type `xs:float`.

Before date, time, or `dateTime` values are compared, they are adjusted to a common timezone. Datetime values without an explicit timezone component use the implicit timezone, which is UTC.



## Returned value

If *sequence-expression* is not the empty sequence, the returned value is the maximum of the values in *sequence-expression*. The data type of the returned value is the same as the data type of the items in *sequence-expression*, or the common data type to which the items in *sequence-expression* are promoted.

If *sequence-expression* is the empty sequence, the empty sequence is returned. If the sequence includes the value NaN, NaN is returned.

## Example

The following function returns the maximum of the sequence (500, 1.0E2, 40.5).

```
fn:max((500, 1.0E2, 40.5))
```

The values are promoted to the xs:double data type. The function returns the xs:double value 5.0E2, which is serialized as "500".

### Related reference

"min function"

---

## min function

The fn:min function returns the minimum of the values in a sequence.

## Syntax

►—fn:min(*sequence-expression*)—◄

### *sequence-expression*

A sequence that contains items of any of the following atomic types, or an empty sequence:

- xs:float
- xs:double
- xs:decimal
- xs:integer
- xs:string
- xs:date
- xs:time
- xs:dateTime
- xdt:untypedAtomic
- xdt:dayTimeDuration
- xdt:yearMonthDuration
- A type that is derived from any of the previously listed types

Input items of type xdt:untypedAtomic are cast to xs:double. After this casting, all of the items in the input sequence must be convertible by promotion or subtype substitution to a common type that supports the **le** operator. The minimum value is computed in this common type. For example, if the input sequence contains items of type money (derived from xs:decimal) and stockprice (derived from xs:float), the minimum is computed in the type xs:float.

Before date, time, or dateTime values are compared, they are adjusted to a common timezone. Datetime values without an explicit timezone component use the implicit timezone, which is UTC.

## Returned value

If *sequence-expression* is not the empty sequence, the returned value is the minimum of the values in *sequence-expression*. The data type of the returned value is the same as the data type of the items in *sequence-expression*, or the common data type to which the items in *sequence-expression* are promoted.

If *sequence-expression* is the empty sequence, the empty sequence is returned. If the sequence includes the value NaN, NaN is returned.

## Examples

**Example with numeric arguments:** The following function returns the minimum of the sequence (500, 1.0E2, 40.5):

```
fn:min((500, 1.0E2, 40.5))
```

The values are promoted to the xs:double data type. The function returns the xs:double value 4.05E1, which is serialized as "40.5".

**Example with string arguments:** The following function returns the minimum of the sequence ("x", "y", "Z") using the default collation. Assume that the default collation sorts lowercase alphabetic characters before uppercase alphabetic characters.

```
fn:min(("x", "y", "Z"))
```

The returned value is "x".

### Related reference

"max function" on page 130

---

## name function

The fn:name function returns the prefix and local name parts of a node name.

### Syntax

```
fn:name(node)
```

*node* The qualified name of a node for which the name is to be retrieved. If *node* is not specified, fn:name is evaluated for the current context node.

### Returned value

The returned value depends on the value of *node*:

- If *node* meets any of the following conditions, a string of length 0 is returned:
  - *node* is the empty sequence.
  - *node* is not an element node, an attribute node, or a processing-instruction node.
- If *node* meets any of the following conditions, an error is returned:
  - *node* is undefined.
  - *node* is not a node.
- Otherwise, an xs:string value is returned that contains the prefix (if present) and local name for *node*.

## Examples

The following query returns the value "comp:emp":

```
declare namespace d="http://www.example.org";
let $department := document {
<comp:dept xmlns:comp="http://www.mycompany.com" id="A07">
  <comp:emp id="31201" />
</comp:dept> }
return fn:name($department/d:dept/d:emp)
```

The following query also returns the value "comp:emp":

```

declare namespace d="http://www.example.org";
let $department := document {
<comp:dept xmlns:comp="http://www.mycompany.com" id="A07">
  <comp:emp id="31201" />
</comp:dept> }
return $department/d:dept/d:emp/fn:name()

```

### Related concepts

“Node properties” on page 7

## namespace-uri function

The fn:namespace-uri function returns the namespace URI of the qualified name for a node.

### Syntax

```

▶▶ fn:namespace-uri( node )

```

*node* The qualified name of a node for which the namespace URI is to be retrieved. If *node* is not specified, fn:namespace-uri is evaluated for the current context node.

### Returned value

The returned value depends on the value of *node*:

- If *node* meets any of the following conditions, a string of length 0 is returned:
  - *node* is the empty sequence.
  - *node* is not an element node or an attribute node.
  - *node* is an element node or an attribute node, but the expanded qualified name for *node* is not in a namespace.
- If *node* meets any of the following conditions, an error is returned:
  - *node* is undefined.
  - *node* is not a node.
- Otherwise, an xs:string value is returned that contains the namespace URI of the expanded name for *node*.

### Examples

The following query returns the value “http://www.mycompany.com”:

```

declare namespace d="http://www.example.org";
let $department := document {
<comp:dept xmlns:comp="http://www.mycompany.com" id="A07">
  <comp:emp id="31201" />
</comp:dept> }
return fn:namespace-uri($department/d:dept/d:emp)

```

The following query also returns the value “http://www.mycompany.com”:

```

declare namespace d="http://www.example.org";
let $department := document {
<comp:dept xmlns:comp="http://www.mycompany.com" id="A07">
  <comp:emp id="31201" />
</comp:dept> }
return $department/d:dept/d:emp/fn:namespace-uri()

```

### Related concepts

“XML namespaces and QNames” on page 12

---

## namespace-uri-for-prefix function

The `fn:namespace-uri-for-prefix` function returns the namespace URI that is associated with a prefix in the in-scope namespaces for an element.

### Syntax

►►—`fn:namespace-uri-for-prefix(prefix,element)`—►►

*prefix* The prefix for which the namespace is returned.

*prefix* has the `xs:string` data type, which can have zero length, or is an empty sequence.

*element*

An element that has an in-scope namespace that is bound to *prefix*.

### Returned value

The returned value depends on the value of *prefix*:

- If *element* has an in-scope namespace whose prefix value matches the value of *prefix*, the namespace URI for that namespace is returned.
- If *element* does not have an in-scope namespace whose prefix value matches the value of *prefix*, the empty sequence is returned.
- If *prefix* is a string of length 0 or is an empty sequence, the namespace URI for the default namespace is returned.

### Example

The following query returns the value `"http://www.mycompany.com"`:

```
declare namespace d="http://www.example.org";
let $department := document {
  <comp:dept xmlns:comp="http://www.mycompany.com" id="A07">
    <comp:emp id="31201" />
  </comp:dept> }
return fn:namespace-uri-for-prefix("comp", $department/d:dept/d:emp)
```

#### Related concepts

“XML namespaces and QNames” on page 12

---

## namespace-uri-from-QName function

The `fn:namespace-uri-from-QName` function returns the namespace URI part of an `xs:QName` value.

### Syntax

►►—`fn:namespace-uri-from-QName(qualified-name)`—►►

*qualified-name*

The qualified name from which the namespace URI part is to be retrieved.

*qualified-name* has the `xs:QName` data type, or is an empty sequence.

### Returned value

If *qualified-name* is not the empty sequence, the value that is returned is an `xs:string` value that is the namespace URI part of *qualified-name*. If *qualified-name* is not in a namespace, a string of length 0 is returned. If *qualified-name* is the empty sequence, the empty sequence is returned.

## Example

This function returns the string value "http://www.mycompany.com":

```
fn:namespace-uri-from-QName(fn:QName("http://www.mycompany.com", "comp:employee"))
```

### Related concepts

"XML namespaces and QNames" on page 12

---

## node-name function

The fn:node-name function returns the expanded QName of a node.

### Syntax

```
fn:node-name(node)
```

*node* The node for which the expanded name is to be retrieved.

### Returned value

The returned value is an xs:QName value that contains the expanded QName for *node*. If *node* is an empty sequence, an empty sequence is returned.

## Example

The following query returns the expanded QName that corresponds to the URI http://www.mycompany.com and the lexical QName comp:emp:

```
declare namespace d="http://www.example.org";
let $department := document {
<comp:dept xmlns:comp="http://www.mycompany.com" id="A07">
  <comp:emp id="31201" />
</comp:dept> }
return fn:node-name($department/d:dept/d:emp)
```

### Related concepts

"Node properties" on page 7

---

## normalize-space function

The fn:normalize-space function strips leading and trailing whitespace characters from a string and replaces each internal sequence of whitespace characters with a single blank character.

### Syntax

```
fn:normalize-space(source-string)
```

#### *source-string*

A string in which whitespace is to be normalized.

*source-string* is an xs:string value or the empty sequence.

If *source-string* is not specified, the argument of fn:normalize-space is the current context item, which is converted to an xs:string value by using the fn:string function.

### Returned value

The returned value is the xs:string value that results when the following operations are performed on *source-string*:

- Leading and trailing whitespace characters are removed.
- Each internal sequence of one or more adjacent whitespace characters is replaced by a single space (X'20') character.

Whitespace characters are space (X'20'), tab (X'09'), line feed (X'0A'), and carriage return (X'0D').

If *source-string* is the empty sequence, a string of length 0 is returned.

## Example

The following function removes extra whitespace characters from the string " a b c d " .

```
fn:normalize-space(" a b c d ")
```

The returned value is "a b c d".

### Related concepts

"Whitespace" on page 14

## normalize-unicode function

The fn:normalize-unicode function performs Unicode normalization on a string.

### Syntax

```
fn:normalize-unicode(source-string [normalization-type])
```

#### *source-string*

A value on which Unicode normalization is to be performed.

*source-string* is an xs:string value or the empty sequence.

#### *normalization-type*

An xs:string value that indicates the type of Unicode normalization that is to be performed.

Possible values are:

**NFC** Unicode Normalization Form C. If *normalization-type* is not specified, NFC normalization is performed.

**NFD** Unicode Normalization Form D.

**NFKC** Unicode Normalization Form KC.

**NFKD** Unicode Normalization Form KD.

If a zero-length string is specified, then no normalization is performed.

### Returned value

If *source-string* is not the empty sequence, the returned value is the xs:string value that results when Unicode normalization that is specified by *normalization-type* is performed on *source-string*. If *normalization-type* is not specified, Unicode Normalization Form C (NFC) is performed on *source-string*. Unicode normalization is described in *Character Model for the World Wide Web 1.0*.

If *source-string* is the empty sequence, a string of length 0 is returned.

## Example

The following function performs Unicode Normalization Form C on the string "ṁ" (a Latin lowercase letter m with a dot below):

```
fn:normalize-unicode("&#x109;&#x803;", "NFC")
```

The returned value is "&x7747;".

---

## not function

The `fn:not` function returns false if the effective boolean value of a sequence is true, and true if the effective boolean value of a sequence is false.

### Syntax

```
fn:not(sequence-expression)
```

*sequence-expression*

Any sequence that contains items of any type, or the empty sequence.

### Returned value

If *sequence-expression* is not an empty sequence, then the value that is returned is true if the effective boolean value of the sequence is false. The returned value is false if the effective boolean value of the sequence is true.

If *sequence-expression* is the empty sequence, the returned value is true.

### Example

The following function returns false because the effective boolean value of a node is true.

```
fn:not(<employee />)
```

#### Related concepts

“Effective Boolean value” on page 51

---

## number function

The `fn:number` function converts a value to the `xs:double` data type.

### Syntax

```
fn:number(atomic-value)
```

*atomic-value*

An atomic value or the empty sequence. If *atomic-value* is not specified, `fn:number` is evaluated for the current context item.

### Returned value

If *atomic-value* is not the empty sequence, the returned value is the result of casting *atomic-value* as `xs:double`. If *atomic-value* cannot be cast to the `xs:double` data type, NaN is returned.

If *numeric-value* is the empty sequence, NaN is returned.

### Example

**Example of converting an `xs:decimal` value to `xs:double`:** The following function converts the `xs:decimal` value 2.75 to `xs:double`.

```
fn:number(2.75)
```

The returned value is 2.75E0 .

**Example of converting an xs:boolean value to xs:double:** The following function converts the boolean value false() to xs:double.

```
fn:number(false())
```

The returned value is 0.0E0.

**Related reference**

“double data type” on page 29

---

## one-or-more function

The fn:one-or-more function returns its argument if the argument contains one or more items.

### Syntax

►►—fn:one-or-more(*sequence-expression*)—►►

*sequence-expression*

Any sequence, including the empty sequence.

### Returned value

If *sequence-expression* contains one or more items, *sequence-expression* is returned. Otherwise, an error is returned.

### Example

The following example uses the fn:one-or-more function to determine if the sequence in variable \$seq contains one or more items.

```
let $seq := (5,10)
return fn:one-or-more($seq)
```

(5,10) is returned.

**Related concepts**

“Sequences and items” on page 4

**Related reference**

“exactly-one function” on page 122

“zero-or-one function” on page 159

---

## position function

The fn:position function returns the position of the context item in the sequence that is currently being processed.

### Syntax

►►—fn:position()—►►

### Returned value

The returned value is an xs:integer value that indicates the position of the context item in the sequence that is currently being processed. If the context item is undefined, an error is returned. The position function returns a deterministic result only if the sequence that contains the context item has a



deterministic order. The position function is typically used in a predicate.

## Example

In the following expression, the position function is called for each item in a sequence of ten items. For each item, the position function returns the position of that item in the sequence. The predicate `position() eq 5` is true only for the fifth item in the sequence.

```
(11 to 20)[position() eq 5]
```

The value returned by the expression is 15.

### Related concepts

“Document order of nodes” on page 10

“Sequences and items” on page 4

### Related reference

“Dynamic context and focus” on page 47

---

## QName function

The `fn:QName` function builds an expanded name from a namespace URI and a string that contains a lexical QName (with an optional prefix.) .

### Syntax

```
fn:QName(URI,QName)
```

*URI* The namespace portion of an expanded name.

*URI* has the `xs:string` data type, or is an empty string or sequence.

*QName*

A value that is the correct lexical form of the data type `xs:QName`.

*QName* has the `xs:string` data type.

### Returned value

The returned value is an `xs:QName` value that is an expanded name with a namespace URI that is specified by *URI*, and the prefix and local name that is specified by *QName*.

The `fn:QName` function associates the namespace prefix of *QName* with the value of *URI*. If *QName* has a namespace prefix, *URI* cannot be a zero-length string or empty sequence. If *QName* has only a local name and no prefix, *URI* can be a zero-length string or empty sequence.

## Example

The following function is given a namespace URI and a string that contains a lexical QName, and it returns a value of type `xs:QName`.

```
fn:QName("http://www.mycompany.com", "comp:employee")
```

The returned value is an `xs:QName` value with namespace URI of "http://www.mycompany.com", a prefix of "comp", and local name of "employee".

### Related concepts

“XML namespaces and QNames” on page 12

---

## remove function

The fn:remove function removes an item from a sequence.

### Syntax

►►—fn:remove(*source-sequence*,*remove-position*)—►►

#### *source-sequence*

The sequence from which an item is to be removed.

*source-sequence* is a sequence of items of any data type, or is the empty sequence.

#### *remove-position*

The position in *source-sequence* of the item that is to be removed. *remove-position* has the xs:integer data type.

### Returned value

If *source-sequence* is not the empty sequence:

- If *remove-position* is less than one or greater than the length of *source-sequence*, the returned value is *source-sequence*.
- If *remove-position* is greater than or equal to one and less than or equal to the length of *source-sequence*, the returned value is a sequence with the following items, in the following order:
  - The items in *source-sequence* before item *remove-position*
  - The items in *source-sequence* after item *remove-position*
- If *source-sequence* is the empty sequence, the returned value is the empty sequence.

### Example

The following function returns the sequence that results from removing the item at position three from the sequence (1,2,4,7):

```
fn:remove((1,2,4,7),3)
```

The returned value is (1,2,7).

#### Related concepts

“Sequences and items” on page 4

---

## replace function

The fn:replace function compares each set of characters within a string to a given pattern, and then it replaces the characters that match the pattern with another set of characters.

### Syntax

►►—fn:replace(*source-string*,*pattern*,*replacement-string*<sub>[,flags]</sub>)—►►

#### *source-string*

A string that contains characters that are to be replaced.

*source-string* is an xs:string value or the empty sequence.

#### *pattern*

A regular expression that is compared to *source-string*. A regular expression is a set of characters, wildcards, and operators that define a string or group of strings in a search pattern.

*pattern* is an xs:string value.

### *replacement-string*

A string that contains characters that replace characters that match *pattern* in *source-string*.

*replacement-string* is an `xs:string` value.

*replacement-string* can contain the variables \$0 to \$9. \$0 represents the entire string in *pattern*. The variable \$1 through \$9 represent one of nine possible parenthesized subexpressions in *pattern*. (\$1 represents the first subexpression, \$2 represents the second subexpression, and so on.)

To use the literal dollar sign (\$) in *replacement-string*, use the string "\$". To use the literal backslash (\) in *replacement-string*, use the string "\\".

*flags* An `xs:string` value that can contain any of the following values that control the matching of *pattern* to *source-string*:

- s** Indicates that the dot (.) replaces any character.  
If the s flag is not specified, the dot (.) replaces any character except the new-line character (X'0A').
- m** Indicates that the caret (^) replaces the start of a line (the position after a new-line character), and the dollar sign (\$) replaces the end of a line (the position before a new-line character).  
If the m flag is not specified, the caret (^) replaces the start of a string, and the dollar sign (\$) replaces the end of the string.
- i** Indicates that matching is case-insensitive.  
If the i flag is not specified, case-sensitive matching is done.
- x** Indicates that whitespace characters within *pattern* are ignored.  
If the x flag is not specified, whitespace characters are used for matching.

## Returned value

If *source-string* is not the empty sequence, the returned value is a string that results when the following operations are performed on *source-string*:

- *source-string* is searched for characters that match *pattern*. If *pattern* contains two or more alternative sets of characters, the first set of characters in *pattern* that matches characters in *source-string* is considered to be the matching pattern.
- Each set of characters in *source-string* that matches *pattern* is replaced with *replacement-string*. If *replacement-string* contains any of the variables \$0 through \$9, the substring of *source-string* that matches the subexpression in *pattern* that corresponds to the variable replaces the variable in *replacement-string*. Then the modified *replacement-string* is inserted into *source-string*. If a variable does not have a corresponding subexpression in *pattern* because there are more variables than subexpressions or a subexpression does not have a match in *source-string*, a string of length 0 replaces the variable in *replacement-string*.

If *pattern* is not found in *source-string*, an error is returned.

If *source-string* is the empty sequence, a string of length 0 is returned.

## Examples

**Example of replacing a substring with another substring:** The following function replaces all instances of "a" in the string "abbcacadbdc" with "ba".

```
fn:replace("abbcacadbdc","a","ba")
```

The returned value is "babbcbacbadbdcd".

**Example of replacing a substring using a replacement string with variables:** The following function replaces "a" and the character that follows "a" with two instances of the character that follows the "a" in "abbcacadbdc".

```
fn:replace("abbcacadbdc", "a(.)", "$1$1")
```

The returned value is "bbcccddbdc".

#### Related reference

"translate function" on page 155

Regular expressions

A regular expression is a sequence of characters that act as a pattern for matching and manipulating strings. Regular expressions are used in the following XQuery functions: `fn:matches`, `fn:replace`, and `fn:tokenize`. DB2 XQuery regular expression support is based on the XML schema regular expression support as defined in the W3C Recommendation *XML Schema Part 2: Datatypes Second Edition* with extensions as defined by W3C Candidate Recommendation *XQuery 1.0 and XPath 2.0 Functions and Operators*.

---

## resolve-QName function

The `fn:resolve-QName` function converts a string containing a lexical QName into an expanded QName by using the in-scope namespaces of an element to resolve the namespace prefix to a namespace URI.

### Syntax

► `fn:resolve-QName(qualified-name, element-for-namespace)` ◄

#### *qualified-name*

A string that is in the form of a qualified name.

*qualified-name* has the `xs:string` data type, or is the empty sequence.

#### *element-for-namespace*

An element that provides the in-scope namespaces for *qualified-name*.

*element-for-namespace* is an element node.

### Returned value

If *qualified-name* is not the empty sequence, the returned value is an expanded name that is constructed as follows:

- The prefix and local name of the expanded QName is taken from *qualified-name*.
- If *qualified-name* has a prefix, and that prefix matches a prefix in the in-scope namespaces of *element-for-namespace*, the namespace URI to which this prefix is bound is the namespace URI for the returned value.
- If *qualified-name* has no prefix, and a default namespace URI is defined in the in-scope namespaces of *element-for-namespace*, this default namespace URI is the namespace URI for the returned value.
- If *qualified-name* has no prefix, and no default namespace URI is defined in the in-scope namespaces of *element-for-namespace*, the returned value has no namespace URI.
- If the prefix for *qualified-name* does not match a namespace prefix in the in-scope namespaces of *element-for-namespace*, or *qualified-name* is not in the form of a valid qualified name, an error is returned.

If *qualified-name* is the empty sequence, the empty sequence is returned.

## Example

The following query returns the expanded QName that corresponds to the URI `http://www.mycompany.com` and the lexical QName `comp:dept`:

```
declare namespace d="http://www.mycompany.com";
let $department := document {
  <comp:dept xmlns:comp="http://www.mycompany.com" id="A07">
    <comp:emp id="31201" />
  </comp:dept> }
return fn:resolve-QName("comp:dept", $department/d:dept/d:emp)
```

### Related concepts

“XML namespaces and QNames” on page 12

---

## reverse function

The `fn:reverse` function reverses the order of the items in a sequence.

### Syntax

►► `fn:reverse(source-sequence)` ◀◀

#### *source-sequence*

The sequence that is to be reversed.

*source-sequence* is a sequence of items of any data type, or is the empty sequence.

### Returned value

If *source-sequence* is not the empty sequence, the returned value is a sequence that contains the items in *source-sequence*, in reverse order.

If *source-sequence* is the empty sequence, the empty sequence is returned.

## Example

The following function returns the items in sequence (1,2,3,7) in reverse order:

```
fn:reverse((1,2,3,7))
```

The returned value is (7,3,2,1).

### Related concepts

“Sequences and items” on page 4

---

## root function

The `fn:root` function returns the root node of a tree to which a node belongs.

### Syntax

►► `fn:root(node)` ◀◀

*node* A node or the empty sequence. The default value for *node* is the context node.

### Returned value

If *node* is not the empty sequence, the returned value is the root node of the tree to which *node* belongs.

If *node* is the root node of the tree, the returned value is *node*.

If *node* is the empty sequence, the returned value is the empty sequence.

## Example

Suppose that some XQuery variables are defined like this:

```
let $f := <first>Laura</first>
let $e := <emp> {$f} <last>Brown</last> </emp>
let $doc := document {<emps>{$e}</emps>}
```

**Example of returning the root node of an element:** The following function returns the root node of the element named last:

```
fn:root($e/last)
```

The returned value is `<emp><first>Laura</first><last>Brown</last></emp>`.

**Example of returning the root node of a document:** The following function returns the root node of the document that is bound to the variable \$doc:

```
fn:root($doc)
```

The returned value is a document node.

### Related concepts

“Node hierarchies” on page 5

---

## round function

The `fn:round` function returns the integer that is closest to a given numeric value.

### Syntax

►—`fn:round(numeric-value)`—►

#### *numeric-value*

An atomic value or an empty sequence.

If *numeric-value* is an atomic value, it has one of the following types:

- `xs:float`
- `xs:double`
- `xs:decimal`
- `xs:integer`
- `xdt:untypedAtomic`
- A type that is derived from any of the previously listed types

If *numeric-value* has the `xdt:untypedAtomic` data type, it is converted to an `xs:double` value.

### Returned value

If *numeric-value* is not the empty sequence, the returned value is the integer that is closest to *numeric-value*. That is, `fn:round(numeric-value)` is equivalent to `fn:floor(numeric-value+0.5)`. The data type of the returned value depends on the data type of *numeric-value*:

- If *numeric-value* is `xs:float`, `xs:double`, `xs:decimal`, or `xs:integer`, the value that is returned has the same type as *numeric-value*.
- If *numeric-value* has a data type that is derived from `xs:float`, `xs:double`, `xs:decimal`, or `xs:integer`, the value that is returned has the direct parent data type of *numeric-value*.

If *numeric-value* is the empty sequence, the returned value is the empty sequence.

## Examples

**Example with a positive argument:** The following function returns the rounded value of 0.5:

```
fn:round(0.5)
```

The returned value is 1.

**Example with a negative argument:** The following function returns the rounded value of (-1.5):

```
fn:round(-1.5)
```

The returned value is -1.

### Related reference

“round-half-to-even function”

---

## round-half-to-even function

The `fn:round-half-to-even` function returns the numeric value with a specified precision that is closest to a given numeric value.

### Syntax

```
fn:round-half-to-even(numeric-value [precision])
```

#### *numeric-value*

An atomic value or an empty sequence.

If *numeric-value* is an atomic value, it has one of the following types:

- `xs:float`
- `xs:double`
- `xs:decimal`
- `xs:integer`
- `xdt:untypedAtomic`
- A type that is derived from any of the previously listed types

If *numeric-value* has the `xdt:untypedAtomic` data type, it is converted to an `xs:double` value.

#### *precision*

The number of digits to the right of the decimal point to which *numeric-value* is to be rounded. *precision* is an `xs:integer` value. The default value for *precision* is 0.

### Returned value

If *numeric-value* is not the empty sequence, and *precision* is 0 or not specified, the returned value is the integer that is closest to *numeric-value*. If *numeric-value* is equally close to two integers, the returned value is the even integer.

If *numeric-value* is not the empty sequence, and *precision* is not 0, the returned value is a numeric value that has *precision* digits to the right of the decimal point and is closest to *numeric-value*. If *numeric-value* is equally close to two values, the returned value is the value whose least significant digit is even.

The data type of the returned value depends on the data type of *numeric-value*:

- If *numeric-value* is `xs:float`, `xs:double`, `xs:decimal`, or `xs:integer`, the value that is returned has the same type as *numeric-value*.
- If *numeric-value* has a data type that is derived from `xs:float`, `xs:double`, `xs:decimal`, or `xs:integer`, the value that is returned has the direct parent data type of *numeric-value*.

If *numeric-value* is the empty sequence, the returned value is the empty sequence.

## Examples

**Example without a precision argument:** The following function returns the rounded value of 0.5:

```
fn:round-half-to-even(0.5)
```

The returned value is 0.

**Example with a non-zero precision argument:** The following function returns 1.5432, rounded to two decimal places.

```
fn:round-half-to-even(1.5432,2)
```

The returned value is 1.54.

**Example with negative precision:** The following function returns 35600.

```
fn:round-half-to-even(35612.25, -2)
```

### Related reference

“round function” on page 144

---

## sqlquery function

The `db2-fn:sqlquery` function retrieves a sequence that is the result of an SQL fullselect in the currently connected DB2 database.

### Syntax

```
db2-fn:sqlquery(string-literal)
```

#### *string-literal*

Contains a fullselect. The fullselect must specify a single-column result set, and the column must have the XML data type.

If the fullselect contains single quotation marks (for example, around a string constant), enclose the function argument in double quotation marks. For example:

```
"select c1 from t1 where c2 = 'Hello'"
```

If the fullselect contains double quotation marks (for example, around a delimited identifier), enclose the function argument in single quotation marks. For example:

```
'select c1 from "t1" where c2 = 47'
```

If the fullselect contains both single and double quotation marks, enclose the function argument in single quotation marks and represent each internal single quote by two adjacent single quote characters. For example:

```
'select c1 from "t1" where c2 = ''Hello'''
```

### Returned value

The returned value is a sequence that is the result of the fullselect in *string-literal*. DB2 processes the fullselect as an SQL statement, following the usual dynamic SQL rules for authorization and name resolution. The XML values that are returned by the fullselect are concatenated to form the result of the function. Rows that contain null values do not affect the result sequence. If the fullselect returns no rows or returns only null values, the result of the function is an empty sequence.



The number of items in the sequence that is returned by the `db2-fn:sqlquery` function can be different from the number of rows that are returned by the `fullselect` because some of these rows can contain null values or sequences with multiple items.

## Examples

**Example of fullselects that return a sequence of documents:** The following example shows several function calls that return the same sequence of documents from table `PRODUCT`. The documents are in column `DESCRIPTION`.

Any of the following functions produce the same result:

```
db2-fn:sqlquery('select description from product')
db2-fn:sqlquery('SELECT DESCRIPTION FROM PRODUCT')
db2-fn:sqlquery('select "DESCRIPTION" from "PRODUCT"')
```

**Example of fullselects that return a single document:** The following example returns a sequence that is a single document in table `PRODUCT`. The document is in column `DESCRIPTION` and is identified by a value of `'100-103-01'` for column `PID`.

Any of the following functions produce the same result:

```
db2-fn:sqlquery('select Description from Product where pID=''100-103-01''')
db2-fn:sqlquery("select description from product where pid='100-103-01'")
db2-fn:sqlquery("select ""DESCRIPTION"" from product where pid='100-103-01'")
```

### Related concepts

“Retrieving DB2 data with XQuery functions” on page 2

### Related reference

“xmlcolumn function” on page 158

---

## starts-with function

The `fn:starts-with` function determines whether a string begins with a given substring.

### Syntax

►► `fn:starts-with(string,substring)` ◀◀

*string* The string to search for *substring*.

*string* has the `xs:string` data type, or is the empty sequence. If *string* is the empty sequence, *string* is set to a string of length 0.

*substring*

The substring to search for at the beginning of *string*.

*substring* has the `xs:string` data type, or is the empty sequence.

### Returned value

The returned value is the `xs:boolean` value `true` if either of the following conditions are satisfied:

- *substring* occurs at the beginning of the *string*.
- *substring* is an empty sequence of a string of length zero.

Otherwise, the returned value is `false`.

## Example

The following function determines whether the string 'Test literal' begins with the string 'lite'.

```
fn:starts-with('Test literal','lite')
```

The returned value is false.

### Related reference

“contains function” on page 114

“ends-with function” on page 121

---

## string function

The fn:string function returns the string representation of a value.

### Syntax

```
►► fn:string(value) ◀◀
```

*value* The value that is to be represented as a string.

*value* is a node or an atomic value, or is the empty sequence.

If *value* is not specified, fn:string is evaluated for the current context item. If the current context item is undefined, an error is returned.

### Returned value

If *value* is not the empty sequence:

- If *value* is a node, the returned value is the string value of the node.
- If *value* is an atomic value, the returned value is the result of casting *value* to the xs:string type.

If *value* is the empty sequence, the result is a string of length 0.

## Example

The following function returns the string representation of 123:

```
fn:string(xs:integer(123))
```

The returned value is '123'.

---

## string-join function

The fn:string-join function returns a string that is generated by concatenating items separated by a separator character.

### Syntax

```
►► fn:string-join(sequence,separator) ◀◀
```

*sequence*

The sequence of items that are to be concatenated to form a string.

*sequence* is any sequence of xs:string values, or an empty sequence.

*separator*

A delimiter that is inserted into the resulting string between items from *sequence*.

*separator* has a data type of `xs:string`.

## Returned value

The returned value is a string that is the concatenation of the items in *sequence*, separated by *separator*. If *separator* is a zero-length string, the items in *sequence* are concatenated without a separator. If *sequence* is an empty sequence, a zero-length string is returned.

## Example

The following function returns the string that is the result of concatenating the items in the sequence ("I", "made", "a", "sentence!"), using the whitespace character as a separator:

```
fn:string-join(("I" , "made", "a", "sentence!"), " ")
```

The returned value is the string "I made a sentence!"

---

## string-length function

The `fn:string-length` function returns the length of a string.

### Syntax

►►—`fn:string-length(source-string)`—►►

#### *source-string*

The string for which the length is to be returned.

*source-string* has the `xs:string` data type, or is an empty sequence.

## Returned value

If *source-string* is not the empty sequence, the returned value is the length of *source-string* in characters. Code points above `xFFFF`, which use two 16-bit values known as a surrogate pairs, are counted as one character in the length of the string. *source-string* is an `xs:integer` value.

If *source-string* is the empty sequence, the returned value is 0.

## Example

The following function returns the length of the string 'Test literal'.

```
fn:string-length('Test literal')
```

The returned value is 12.

---

## string-to-codepoints function

The `fn:string-to-codepoints` function returns a sequence of Unicode code points that correspond to a string value.

### Syntax

►►—`fn:string-to-codepoints(source-string)`—►►

#### *source-string*

A string value for which the Unicode code point for each character is to be returned, or the empty sequence.

## Returned value

If *source-string* is not the empty sequence and does not have length 0, the returned value is a sequence of xs:integer values that represent the code points for the characters in *source-string*.

If *source-string* is the empty sequence or has length 0, the returned value is the empty sequence.

## Example

The following function returns a sequence of code points that represent the characters in the string 'XQuery'.

```
fn:string-to-codepoints("XQuery")
```

The returned value is (88,81,117,101,114,121).

### Related reference

“codepoints-to-string function” on page 112

---

## subsequence function

The fn:subsequence function returns a subsequence of a sequence.

## Syntax

```
fn:subsequence(source-sequence, start , length)
```

### *source-sequence*

The sequence from which the subsequence is retrieved.

*source-sequence* is any sequence, including the empty sequence.

*start* The starting position in *source-sequence* of the subsequence. The first position of *source-sequence* is 1. If  $start \leq 0$ , *start* is set to 1.

*start* has the xs:double data type.

*length* The number of items in the subsequence. The default for *length* is the number of items in *source-sequence*. If  $start + length - 1$  is greater than the length of *source-sequence*, *length* is set to  $(\text{length of } source\text{-sequence}) - start + 1$ .

*length* has the xs:double data type.

## Returned value

If *source-sequence* is not the empty sequence, the returned value is a subsequence of *source-sequence* that starts at position *start* and contains *length* items.

If *source-sequence* is the empty sequence, the empty sequence is returned.

## Example

The following function returns three items from the sequence ('T','e','s','t',' ','s','e','q','u','e','n','c','e'), starting at the sixth item.

```
fn:subsequence(('T','e','s','t',' ','s','e','q','u','e','n','c','e'),6,3)
```

The returned value is ('s','e','q').

### Related concepts

“Sequences and items” on page 4

---

## substring function

The `fn:substring` function returns a substring of a string.

### Syntax

► `fn:substring(source-string,start [,length])` ◄

#### *source-string*

The string from which the substring is retrieved.

*source-string* has the `xs:string` data type, or is an empty sequence.

*start* The starting character position in *source-string* of the substring. The first position of *source-string* is 1. If *start* ≤ 0, *start* is set to 1. Code points above xFFFF, which use two 16-bit values known as a surrogate pairs, are counted as one character.

*start* has the `xs:double` data type.

*length* The length in characters of the substring. The default for *length* is the length of *source-string*. If *start+length-1* is greater than the length of *source-string*, *length* is set to (length of *source-string*)-*start*+1. Code points above xFFFF, which use two 16-bit values known as a surrogate pairs, are counted as one character in the length of the string.

*length* has the `xs:double` data type.

### Returned value

If *source-string* is not the empty sequence, the returned value is a substring of *source-string* that starts at character position *start* and has *length* characters. If *source-string* is the empty sequence, the result is a string of length 0.

### Example

The following function returns seven characters starting at the sixth character of the string 'Test literal'.

```
fn:substring('Test literal',6,7)
```

The returned value is 'literal'.

#### Related reference

“substring-after function”

“substring-before function” on page 152

---

## substring-after function

The `fn:substring-after` function returns a substring that occurs in a string after the end of the first occurrence of a given search string.

### Syntax

► `fn:substring-after(source-string,search-string)` ◄

#### *source-string*

The string from which the substring is retrieved.

*source-string* has the `xs:string` data type, or is an empty sequence. If *source-string* is the empty sequence, *source-string* is set to a string of length 0.

*search-string*

The string whose first occurrence in *source-string* is to be searched for.

*search-string* has the xs:string data type, or is an empty sequence.

## Returned value

If *source-string* is not the empty sequence or a string of length 0:

- Suppose that the length of *source-string* is  $n$ , and  $m < n$ . If *search-string* is found in *source-string*, and the end of the first occurrence of *search-string* in *source-string* is at position  $m$ , the returned value is the substring that begins at position  $m+1$ , and ends at position  $n$  of *source-string*.
- Suppose that the length of *source-string* is  $n$ . If *search-string* is found in *source-string*, and the end of the first occurrence of *search-string* in *source-string* is at position  $n$ , the returned value is a string of length 0.
- If *search-string* is the empty string or a string of length 0, the returned value is *source-string*.
- If *search-string* is not found in *source-string*, the returned value is a string of length 0.

If *source-string* is the empty sequence or a string of length 0, the returned value is a string of length 0.

## Example

The following function finds the characters after 'ABC' in string to 'DEFABCD' using the default collation.

```
fn:substring-after('DEFABCD', 'ABC')
```

The returned value is 'D'.

### Related reference

“substring-before function”

“substring function” on page 151

---

## substring-before function

The fn:substring-before function returns a substring that occurs in a string before the first occurrence of a given search string.

## Syntax

►►—fn:substring-before(*source-string*,*search-string*)—►►

*source-string*

The string from which the substring is retrieved.

*source-string* has the xs:string data type, or is an empty sequence. If *source-string* is an empty sequence, *source-string* is set to a string of length 0.

*search-string*

The string whose first occurrence in *source-string* is to be searched for.

*search-string* has the xs:string data type, or is an empty sequence.

## Returned value

If *source-string* is not the empty sequence or a string of length 0:

- If *search-string* is found at position  $m$  of *source-string*, and  $m > 1$ , the returned value is the substring that begins at position 1, and ends at position  $m$  of *source-string*.
- If *search-string* is found at position 1 of *source-string*, the returned value is a string of length 0.
- If *search-string* is an empty sequence or a string of length 0, the returned value is *source-string*.

- If *search-string* is not found in *source-string*, the returned value is a string of length 0.

If *source-string* is the empty sequence or a string of length 0, the returned value is a string of length 0.

## Example

The following function finds the characters before 'ABC' in string to 'DEFABCD' using the default collation.

```
fn:substring-before('DEFABCD', 'ABC')
```

The returned value is 'DEF'.

### Related reference

“substring-after function” on page 151

“substring function” on page 151

## sum function

The fn:sum function returns the sum of the values in a sequence.

### Syntax

```
fn:sum(sequence-expression [,empty-sequence-replacement])
```

#### *sequence-expression*

A sequence that contains items of any of the following atomic types, or an empty sequence:

- xs:float
- xs:double
- xs:decimal
- xs:integer
- xdt:untypedAtomic
- xdt:dayTimeDuration
- xdt:yearMonthDuration
- A type that is derived from any of the previously listed types

Input items of type xdt:untypedAtomic are cast to xs:double. After this casting, all of the items in the input sequence must be convertible to a common type by promotion or subtype substitution.

The sum is computed in this common type. For example, if the input sequence contains items of type money (derived from xs:decimal) and stockprice (derived from xs:float), the sum is computed in the type xs:float.

#### *empty-sequence-replacement*

The value that is returned if *sequence-expression* is the empty sequence. *empty-sequence-replacement* can have one of the data types that is listed for *sequence-expression*.

## Returned value

If *sequence-expression* is not the empty sequence, the returned value is the sum of the values in *sequence-expression*. The data type of the returned value is the same as the data type of the items in *sequence-expression*, or the data type to which the items in *sequence-expression* are promoted.

If *sequence-expression* is the empty sequence, and *empty-sequence-replacement* is not specified, fn:sum returns 0.0E0. If *sequence-expression* is an empty sequence, and *empty-sequence-replacement* is specified, fn:sum returns *empty-sequence-replacement*.

## Example

The following function returns the sum of the sequence (500, 1.0E2, 40.5):

```
fn:sum((500, 1.0E2, 40.5))
```

The values are promoted to the xs:double data type. The function returns the xs:double value 6.405E2, which is serialized as "640.5".

### Related concepts

"Sequences and items" on page 4

---

## tokenize function

The fn:tokenize function breaks a string into a sequence of substrings.

### Syntax

```
fn:tokenize(source-string, pattern [flags])
```

#### *source-string*

A string that is to be broken into a sequence of substrings.

*source-string* is an xs:string value or the empty sequence.

#### *pattern*

The delimiter between substrings in *source-string*.

*pattern* is an xs:string value that contains a regular expression. A regular expression is a set of characters, wildcards, and operators that define a string or group of strings in a search pattern.

*flags* An xs:string value that can contain any of the following values that control how *pattern* is matched to characters in *source-string*:

- s** Indicates that the dot (.) in the regular expression matches any character, including the new-line character (X'0A').  
If the s flag is not specified, the dot (.) matches any character except the new-line character (X'0A').
- m** Indicates that the caret (^) matches the start of a line (the position after a new-line character), and the dollar sign (\$) matches the end of a line (the position before a new-line character).  
If the m flag is not specified, the caret (^) matches the start of a string, and the dollar sign (\$) matches the end of the string.
- i** Indicates that matching is case-insensitive.  
If the i flag is not specified, case-sensitive matching is done.
- x** Indicates that whitespace characters within *pattern* are ignored.  
If the x flag is not specified, whitespace characters are used for matching.

### Returned value

If *source-string* is not the empty sequence or a zero-length string, the returned value is a sequence that results when the following operations are performed on *source-string*:

- *source-string* is searched for characters that match *pattern*.
- If *pattern* contains two or more alternative sets of characters, the first set of characters in *pattern* that matches characters in *source-string* is considered to be the matching pattern.



- Each set of characters that does not match *pattern* becomes an item in the result sequence.
- If *pattern* matches characters at the beginning of *source-string*, the first item in the returned sequence is a string of length 0.
- If two successive matches for *pattern* are found within *source-string*, a string of length 0 is added to the sequence.
- If *pattern* matches characters at the end of *source-string*, the last item in the returned sequence is a string of length 0.

If *pattern* is not found in *source-string*, an error is returned.

If *source-string* is the empty sequence, or is the zero-length string, the result is the empty sequence.

## Example

The following function creates a sequence from the string "Tokenize this sentence, please." "\s+" is a regular expression that denotes one or more whitespace characters.

```
fn:tokenize("Tokenize this sentence, please.", "\s+")
```

The returned value is the sequence ("Tokenize", "this", "sentence,", "please.").

### Related reference

"concat function" on page 114

Regular expressions

A regular expression is a sequence of characters that act as a pattern for matching and manipulating strings. Regular expressions are used in the following XQuery functions: fn:matches, fn:replace, and fn:tokenize. DB2 XQuery regular expression support is based on the XML schema regular expression support as defined in the W3C Recommendation *XML Schema Part 2: Datatypes Second Edition* with extensions as defined by W3C Candidate Recommendation *XQuery 1.0 and XPath 2.0 Functions and Operators*.

---

## translate function

The fn:translate function replaces selected characters in a string with replacement characters.

### Syntax

►►—fn:translate(*source-string*,*original-string*,*replacement-string*)—►►

#### *source-string*

The string in which characters are to be converted.

*source-string* has the xs:string data type, or is the empty sequence.

#### *original-string*

A string that contains the characters that can be converted.

*original-string* has the xs:string data type.

#### *replacement-string*

A string that contains the characters that replace the characters in *original-string*.

*replacement-string* has the xs:string data type.

If the length of *replacement-string* is greater than the length of *original-string*, the additional characters in *replacement-string* are ignored.

## Returned value

If *source-string* is not the empty sequence, the returned value is the xs:string value that results when the following operations are performed:

- For each character in *source-string* that appears in *original-string*, replace the character in *source-string* with the character in *replacement-string* that appears at the same position as the character in *original-string*.

If the length of *original-string* is greater than the length of *replacement-string*, delete each character in *source-string* that appears in *original-string*, but the character position in *original-string* does not have a corresponding position in *replacement-string*.

If a character appears more than once in *original-string*, the position of the first occurrence of the character in *original-string* determines the character in *replacement-string* that is used.

- For each character in *source-string* that does not appear in *original-string*, leave the character as it is.

If *source-string* is the empty sequence, a string of length 0 is returned.

## Examples

The following function returns the string that results from replacing e with o and l with m in the string 'Test literal'.

```
fn:translate('Test literal','el','om')
```

The returned value is 'Tost mitoram'.

The following function returns the string that results from replacing A with B, t with f, e with i, and r with m in the string literal 'Another test literal'.

```
fn:translate('Another test literal', 'Ater', 'Bfim')
```

The returned value is 'Bnofhim fisf lifimal'.

### Related reference

“replace function” on page 140

---

## true function

The fn:true function returns the xs:boolean value true.

## Syntax

►►—fn:true()—◀◀

## Returned value

The returned value is the xs:boolean value true.

## Example

Use the true function to return the value true.

```
fn:true()
```

The value true is returned.

### Related reference

“false function” on page 123

---

## unordered function

The `fn:unordered` function returns the items in a sequence in non-deterministic order.

### Syntax

►► `fn:unordered(sequence-expression)` ◀◀

*sequence-expression*

Any sequence, including the empty sequence.

### Returned value

The returned value is the items in *sequence-expression* in non-deterministic order. This assists the query optimizer in choosing access paths that are not dependent on the order of the items in the sequence.

### Example

The following function returns the items in sequence (1,2,3) in non-deterministic order.

```
fn:unordered((1,2,3))
```

#### Related concepts

“Sequences and items” on page 4

#### Related reference

“Order of results in XQuery expressions” on page 48

---

## upper-case function

The `fn:upper-case` function converts a string to uppercase.

### Syntax

►► `fn:upper-case(source-string)` ◀◀

*source-string*

The string that is to be converted to uppercase.

*source-string* has the `xs:string` data type, or is an empty sequence.

### Returned value

If *source-string* is not the empty sequence, the returned value is *source-string*, with each character converted to uppercase.

If *source-string* is not an empty sequence, the returned value is *source-string*, with each character converted to its upper-case correspondent as defined in the Unicode standard. Every character that does not have an upper-case correspondent is included in the returned value in its original form

If *source-string* is the empty sequence, the returned value is a string of length 0.

### Example

The following function converts the string 'Test literal 1' to uppercase.

```
fn:upper-case('Test literal 1')
```

The returned value is 'TEST LITERAL 1'.

#### Related reference

---

## xmlcolumn function

The db2-fn:xmlcolumn function retrieves a sequence from a column in the currently connected DB2 database.

### Syntax

►—db2-fn:xmlcolumn(*string-literal*)—◄

#### *string-literal*

Specifies the name of the column from which the sequence is retrieved. The column name must be qualified by a table name, view name, or alias name, and it must reference a column with the XML data type. The SQL schema name is optional. If you do not specify the SQL schema name, the CURRENT SCHEMA special register is used as the implicit qualifier for the table or view. The *string-literal* is case sensitive. *string-literal* must use the exact characters that identify the column name in the database.

### Returned value

The returned value is a sequence that is the concatenation of the non-null XML values in the column that is specified by *string-literal*. If there are no rows in the table or view, db2-fn:xmlcolumn returns the empty sequence.

The number of items in the sequence that is returned by the db2-fn:xmlcolumn function can be different from the number of rows in the specified table or view because some of these rows can contain null values or sequences with multiple items.

The db2-fn:xmlcolumn function is related to the db2-fn:sqlquery function, and both can produce the same result. However, the arguments of the two functions differ in case sensitivity. The argument in the db2-fn:xmlcolumn function is processed by XQuery, and so it is case sensitive. Because table names and column names in DB2 are in upper-case by default, the argument of db2-fn:xmlcolumn is usually in upper-case. The argument of the db2-fn:sqlquery function is processed by SQL, which automatically converts identifiers to upper-case.

The following function calls are equivalent and return the same results:

```
db2-fn:xmlcolumn('SQLSCHEMA.TABLENAME.COLNAME')
db2-fn:sqlquery('select colname from sqlschema.tablename')
```

### Examples

**Example that returns a sequence of documents:** The following function returns a sequence of XML documents that are stored in the XML column DESCRIPTION in the table named PRODUCT, which, for this example, is in the SQL schema SAMPLE.

```
db2-fn:xmlcolumn('SAMPLE.PRODUCT.DESCRPTION')
```

**Example that uses an implicit SQL schema:** In the following example, the CURRENT SCHEMA special register in DB2 is set to SAMPLE, and so the function returns the same results as the previous example:

```
db2-fn:xmlcolumn('PRODUCT.DESCRPTION')
```

**Example that uses an SQL delimited identifier:** The following function returns a sequence of documents that are stored in the "Thesis" column of the "Student" table, assuming that the table is in the schema currently assigned to CURRENT SCHEMA. Because the table name and column name contain lower-case characters, there are two ways that they can be specified in the string literal argument of the db2-fn:xmlcolumn function:

- Specified as SQL-delimited identifiers (enclosed in double quotes):  
db2-fn:xmlcolumn("Student"."Thesis")
- Specified as a string without indication that they are SQL-delimited identifiers:  
db2-fn:xmlcolumn('Student.Thesis')

By contrast, the same table and column information that is used in the db2-fn:sqlquery function is required to use the SQL-delimited identifiers as follows:

```
db2-fn:sqlquery('select "Thesis" from "Student"')
```

#### Related concepts

“Retrieving DB2 data with XQuery functions” on page 2

#### Related reference

“sqlquery function” on page 146

## zero-or-one function

The fn:zero-or-one function returns its argument if the argument contains one item or is the empty sequence.

### Syntax

►—fn:zero-or-one(*sequence-expression*)—◀

*sequence-expression*

Any sequence, including the empty sequence.

### Returned value

If *sequence-expression* contains one item or is the empty sequence, *sequence-expression* is returned. Otherwise, an error is returned.

### Example

The following example uses the fn:zero-or-one function to determine if the sequence in variable \$seq contains one or fewer items.

```
let $seq := (5,10)
return fn:zero-or-one($seq)
```

An error is returned because the sequence contains two items.

#### Related concepts

“Sequences and items” on page 4

#### Related reference

“exactly-one function” on page 122

“one-or-more function” on page 138



## Chapter 6. Limits

DB2 XQuery has size limits and limits for data types.

### Limits for XQuery data types

This topic identifies the range of values that are allowed for specific DB2 XQuery data types.

Table 34. Limits for XQuery numeric data types

Data type	Description	Limit
xs:float	Smallest value	-3.4028234663852886e+38
	Largest value	+3.4028234663852886e+38
	Smallest positive value	+1.1754943508222875e-38
	Largest negative value	-1.1754943508222875e-38
xs:double	Smallest value	-1.7976931348623158e+308
	Largest value	+1.7976931348623158e+308
	Smallest positive value	+2.2250738585072014e-308
	Largest negative value	+2.2250738585072014e-308
xs:decimal	Largest decimal precision	31 digits
xs:integer	Smallest value	-9 223 372 036 854 775 808
	Largest value	+9 223 372 036 854 775 807
xs:nonPositiveInteger	Smallest value	-9 223 372 036 854 775 808
	Largest value	0
xs:negativeInteger	Smallest value	-9 223 372 036 854 775 808
	Largest value	-1
xs:long	Smallest value	-9 223 372 036 854 775 808
	Largest value	9 223 372 036 854 775 807
xs:int	Smallest value	-2 147 483 648
	Largest value	+2 147 483 647
xs:short	Smallest value	-32 768
	Largest value	+32 767
xs:byte	Smallest value	-128
	Largest value	+127
xs:nonNegativeInteger	Smallest value	0
	Largest value	+9 223 372 036 854 775 807
xs:unsignedLong	Smallest value	0
	Largest value	+9 223 372 036 854 775 807
xs:unsignedInt	Smallest value	0
	Largest value	4 294 967 295
xs:unsignedShort	Smallest value	0
	Largest value	+65 535
xs:unsignedByte	Smallest value	0
	Largest value	+255

Table 34. Limits for XQuery numeric data types (continued)

Data type	Description	Limit
xs:positiveInteger	Smallest value	+1
	Largest value	+9 223 372 036 854 775 807

Table 35. Limits for XQuery date, time, and duration data types

Data type	Description	Limit
xs:duration	Smallest value	-P8333333333333333Y3M11574074074DT1H46M39.999999S
	Largest value	P8333333333333333Y3M11574074074DT1H46M39.999999S
xdt:yearMonthDuration	Smallest value	-P8333333333333333Y3M
	Largest value	P8333333333333333Y3M
xdt:dayTimeDuration	Smallest value	-P11574074074DT1H46M39.999999S
	Largest value	P11574074074DT1H46M39.999999S
xs:dateTime <sup>1, 2</sup>	Smallest value	0001-01-01T00:00:00.000000Z
	Largest value	9999-12-31T23:59:59.999999Z
xs:date <sup>1</sup>	Smallest value	0001-01-01Z
	Largest value	9999-12-31Z
xs:time <sup>2</sup>	Smallest value	00:00:00Z
	Largest value	23:59:59Z
xs:gDay <sup>1</sup>	Smallest value	01Z
	Largest value	31Z
xs:gMonth <sup>1</sup>	Smallest value	01Z
	Largest value	12Z
xs:gYear <sup>1</sup>	Smallest value	0001Z
	Largest value	9999Z
xs:gYearMonth <sup>1</sup>	Smallest value	0001-01Z
	Largest value	9999-12Z
xs:gMonthDay <sup>1</sup>	Smallest value	01-01Z
	Largest value	12-31Z

**Note:** DB2 XQuery provides no support for negative dates.

### Related concepts

Chapter 2, “Type system,” on page 17

### Related reference

“Type casting” on page 23

## Size limits

DB2 XQuery has size limits for string literals and queries.

The size limit for a string literal is 32672 bytes.

The size limit for the length of a query is 2 097 152 bytes.

### Related reference

“Literals” on page 52



## DB2 technical library in PDF format

The following tables describe the DB2 library available from the IBM® Publications Center at [www.ibm.com/shop/publications/order](http://www.ibm.com/shop/publications/order).

Although the tables identify books available in print, the books might not be available in your country or region.

The information in these books is fundamental to all DB2 users; you will find this information useful whether you are a programmer, a database administrator, or someone who works with DB2 Connect™ or other DB2 products.

Table 36. DB2 technical information

Name	Form Number	Available in print
Administration Guide: Implementation	SC10-4221	Yes
Administration Guide: Planning	SC10-4223	Yes
Administrative API Reference	SC10-4231	Yes
Administrative SQL Routines and Views	SC10-4293	No
Call Level Interface Guide and Reference, Volume 1	SC10-4224	Yes
Call Level Interface Guide and Reference, Volume 2	SC10-4225	Yes
Command Reference	SC10-4226	No
Data Movement Utilities Guide and Reference	SC10-4227	Yes
Data Recovery and High Availability Guide and Reference	SC10-4228	Yes
Developing ADO.NET and OLE DB Applications	SC10-4230	Yes
Developing Embedded SQL Applications	SC10-4232	Yes
Developing Java™ Applications	SC10-4233	Yes
Developing Perl and PHP Applications	SC10-4234	No
Getting Started with Database Application Development	SC10-4252	Yes
Getting started with DB2 installation and administration on Linux® and Windows®	GC10-4247	Yes
Message Reference Volume 1	SC10-4238	No
Message Reference Volume 2	SC10-4239	No
Migration Guide	GC10-4237	Yes
Net Search Extender Administration and User's Guide <b>Note:</b> HTML for this document is not installed from the HTML documentation CD.	SH12-6842	Yes
Performance Guide	SC10-4222	Yes

Table 36. DB2 technical information (continued)

Name	Form Number	Available in print
Query Patroller Administration and User's Guide	GC10-4241	Yes
Quick Beginnings for DB2 Clients	GC10-4242	No
Quick Beginnings for DB2 Servers	GC10-4246	Yes
Spatial Extender and Geodetic Data Management Feature User's Guide and Reference	SC18-9749	Yes
SQL Guide	SC10-4248	Yes
SQL Reference, Volume 1	SC10-4249	Yes
SQL Reference, Volume 2	SC10-4250	Yes
System Monitor Guide and Reference	SC10-4251	Yes
Troubleshooting Guide	GC10-4240	No
Visual Explain Tutorial	SC10-4319	No
What's New	SC10-4253	Yes
XML Extender Administration and Programming	SC18-9750	Yes
XML Guide	SC10-4254	Yes
XQuery Reference	SC18-9796	Yes

Table 37. Technical information specific to DB2 Connect

Name	Form Number	Available in print
DB2 Connect User's Guide	SC10-4229	Yes
Quick Beginnings for DB2 Connect Personal Edition	GC10-4244	Yes
Quick Beginnings for DB2 Connect Servers	GC10-4243	Yes

Table 38. WebSphere Information Integration technical information

Name	Form Number	Available in print
WebSphere® Information Integration: Administration Guide for Federated Systems	SC19-1001	Yes
WebSphere Information Integration: ASNCLP Program Reference for Replication and Event Publishing	SC19-1000	Yes
WebSphere Information Integration: Configuration Guide for Federated Data Sources	No form number	No
WebSphere Information Integration: SQL Replication Guide and Reference	SC19-1002	Yes

**Note:** The DB2 Release Notes provide additional information specific to your product's release and fix pack level.

---

## Ordering printed DB2 books

If you require printed DB2 books, you can buy them online in many but not all countries or regions.

You can always order printed DB2 books from your local IBM representative. Keep in mind that some softcopy books on the DB2 PDF Documentation CD are unavailable in print. For example, neither volume of the DB2 Message Reference is available as a printed book.

Printed versions of many of the DB2 books available on the DB2 PDF Documentation CD can be ordered for a fee from IBM. Depending on where you are placing your order from, you may be able to order books online, from the IBM Publications Center. If online ordering is not available in your country or region, you can always order printed DB2 books from your local IBM representative. Note that not all books on the DB2 PDF Documentation CD are available in print.

**Note:** The most up-to-date and complete DB2 documentation is maintained in the DB2 Information Center at <http://publib.boulder.ibm.com/infocenter/db2help/>.

To order printed DB2 books:

1. To find out whether you can order printed DB2 books online in your country or region, check the IBM Publications Center at <http://www.ibm.com/shop/publications/order>. You must select a country, region, or language to access publication ordering information and then follow the ordering instructions for your location.
2. To order printed DB2 books from your local IBM representative, locate the contact information for your local representative from one of the following Web sites:
  - a. The IBM directory of world wide contacts at <http://www.ibm.com/planetwide>.
  - b. The IBM Publications Web site at <http://www.ibm.com/shop/publications/order>. You will need to select your country, region, or language to the access appropriate publications home page for your location. From this page, follow the "About this site" link.
3. When you call, specify that you want to order a DB2 publication.
4. Provide your representative with the titles and form numbers of the books that you want to order.



---

## DB2 troubleshooting information

A wide variety of troubleshooting and problem determination information is available to assist you in using DB2 products.

The following resources are available:

### **DB2 documentation**

Troubleshooting information can be found in the DB2 Troubleshooting Guide or the Support and Troubleshooting section of the DB2 Information Center. There you will find information on how to isolate and identify problems using DB2 diagnostic tools and utilities, solutions to some of the most common problems, and other advice on how to solve problems you might encounter with your DB2 products.

### **DB2 Technical Support Web site**

Refer to the DB2 Technical Support Web site if you are experiencing problems and want help finding possible causes and solutions. The Technical Support site has links to the latest DB2 publications, TechNotes, Authorized Program Analysis Reports (APARs or bug fixes), fix packs, and other resources. You can search through this knowledge base to find possible solutions to your problems.

Access the DB2 Technical Support Web site at <http://www.ibm.com/software/data/db2/udb/support.html>.



---

## Notices

IBM may not offer the products, services, or features discussed in this document in all countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing  
IBM Corporation  
North Castle Drive  
Armonk, NY 10504-1785  
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country/region or send inquiries, in writing, to:

IBM World Trade Asia Corporation  
Licensing  
2-31 Roppongi 3-chome, Minato-ku  
Tokyo 106, Japan

**The following paragraph does not apply to the United Kingdom or any other country/region where such provisions are inconsistent with local law:** INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product, and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information that has been exchanged, should contact:

IBM Canada Limited  
Office of the Lab Director  
8200 Warden Avenue

Markham, Ontario  
L6G 1C7  
CANADA

Such information may be available, subject to appropriate terms and conditions, including in some cases payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems, and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements, or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility, or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information may contain examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious, and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

#### COPYRIGHT LICENSE:

This information may contain sample application programs, in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. \_ enter the year or years\_. All rights reserved.

#### **Trademarks**

Company, product, or service names identified in the documents of the DB2 Version 9 documentation library may be trademarks or service marks of International Business Machines Corporation or other companies. Information on the trademarks of IBM Corporation in the United States, other countries, or both is located at <http://www.ibm.com/legal/copytrade.shtml>.



The following terms are trademarks or registered trademarks of other companies and have been used in at least one of the documents in the DB2 documentation library:

Microsoft<sup>®</sup>, Windows, Windows NT<sup>®</sup>, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel<sup>®</sup>, Itanium<sup>®</sup>, Pentium<sup>®</sup>, and Xeon<sup>™</sup> are trademarks of Intel Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX<sup>®</sup> is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.



---

## Contacting IBM

To contact IBM in your country or region, check the IBM Directory of Worldwide Contacts at <http://www.ibm.com/planetwide>.

To learn more about DB2 products, go to <http://www.ibm.com/software/data/db2/>.



---

# Index

## A

- abbreviated syntax 61
- abs function 109
- and operator 73
- anyAtomicType data type 25
- anySimpleType data type 25
- anyType data type 26
- anyURI data type 26
- arithmetic expressions 67
- atomic types 17
- atomic values 5
- atomization 50
- attribute axis 59
- attribute nodes 8
- attributes
  - computed constructors
    - description 84
  - constructing 84
  - namespace declaration 79
- avg function 110
- axes
  - abbreviated syntax 61
  - in path expressions 59
- axis steps
  - in path expressions 58
  - node tests 59

## B

- base64Binary data type 26
- boolean data type 26
- boolean function 111
- boolean functions, list of 106
- boundary whitespace
  - declaration 40
  - in direct element constructors 80
- boundary-space declarations 40
- built-in data types, constructors for 22
- built-in functions 105
- byte data type 26

## C

- case sensitivity, query language 14
- cast expressions 102
- casting, data types 23
- ceiling function 112
- character references 54
- child axis 59
- codepoints-to-string function 112
- comment nodes 9
- comments
  - computed constructors 88
  - constructing, overview 88
  - direct constructors 88
  - query language, description 15
- compare function 113

- comparison expressions
  - general 71
  - nodes 73
  - overview 69
  - value 69
- computed attribute constructors 84
- computed constructors
  - attribute
    - description 84
    - comment 88
    - description 75
    - element 83
    - processing instruction 87
- concat function 114
- conditional expressions 100
- construction declarations 41
- constructors
  - attribute
    - description 84
  - built-in types 22
  - computed attribute 84
  - computed comment 88
  - computed element 83
  - computed processing instruction 87
  - direct comment 88
  - direct element
    - description 76
  - direct processing instruction 87
  - document node 85
  - enclosed expressions 75
  - in-scope namespaces 82
  - namespace declaration attributes 79
  - processing instruction 86
  - text node 86
  - XML 75
- contains function 114
- context item expressions 55
- context of expressions 47
- copy-namespace declarations 41
- count function 115
- current-date function 115
- current-dateTime function 116
- current-time function 116

## D

- data function 117
- data model
  - XQuery and XPath 4
- data type hierarchy 17
- data types
  - built-in, constructors for 22
  - casting 23
  - categories 19
  - conversions and matching 47
  - date, time, and duration, list of 20
  - generic, list of 19
  - limits for 161

- data types *(continued)*
  - lists of 19
  - numeric, list of 20
  - other, list of 21
  - overview 17
  - promotion 51
  - string, list of 19
  - substitution 50
  - untyped, list of 19
  - xd<sub>t</sub>:anyAtomicType 25
  - xd<sub>t</sub>:dayTimeDuration 28
  - xd<sub>t</sub>:untyped 37
  - xd<sub>t</sub>:untypedAtomic 37
  - xd<sub>t</sub>:yearMonthDuration 38
  - xs:anySimpleType 25
  - xs:anyType 26
  - xs:anyURI 26
  - xs:base64Binary 26
  - xs:boolean 26
  - xs:byte 26
  - xs:date 26
  - xs:dateTime 27
  - xs:decimal 29
  - xs:double 29
  - xs:duration 30
  - xs:ENTITY 31
  - xs:float 31
  - xs:gDay 31
  - xs:gMonth 32
  - xs:gMonthDay 32
  - xs:gYear 33
  - xs:gYearMonth 33
  - xs:hexBinary 33
  - xs:ID 33
  - xs:IDREF 33
  - xs:int 34
  - xs:integer 34
  - xs:language 34
  - xs:long 34
  - xs:Name 34
  - xs:NCName 34
  - xs:negativeInteger 34
  - xs:NMTOKEN 35
  - xs:nonNegativeInteger 35
  - xs:nonPositiveInteger 35
  - xs:normalizedString 35
  - xs:NOTATION 35
  - xs:positiveInteger 35
  - xs:QName 36
  - xs:short 36
  - xs:string 36
  - xs:time 36
  - xs:token 37
  - xs:unsignedByte 37
  - xs:unsignedInt 37
  - xs:unsignedLong 37
  - xs:unsignedShort 37
- date data type 26
- date data types, list of 20
- date functions, list of 107
- dateTime data type 27
- dateTime function 117
- dayTimeDuration data type 28
- DB2 XQuery functions
  - abs 109
  - avg 110
  - boolean 111
  - boolean functions, list of 106
  - ceiling 112
  - codepoints-to-string 112
  - compare 113
  - concat 114
  - contains 114
  - count 115
  - current-date 115
  - current-dateTime 116
  - current-time 116
  - data 117
  - date functions, list of 107
  - dateTime 117
  - deep-equal 118
  - default-collation 120
  - distinct-values 120
  - empty 121
  - ends-with 121
  - exactly-one 122
  - exists 122
  - false 123
  - floor 123
  - implicit-timezone function 124
  - in-scope-prefixes 124
  - index-of 125
  - insert-before 126
  - last 127
  - local-name 127
  - local-name-from-QName 128
  - lower-case 128
  - matches 129
  - max 130
  - min 131
  - name 132
  - namespace-uri 133
  - namespace-uri-for-prefix 134
  - namespace-uri-from-QName 134
  - node functions, list of 108
  - node-name 135
  - normalize-space 135
  - normalize-unicode 136
  - not 137
  - number 137
  - number functions, list of 106
  - one-or-more 138
  - other functions, list of 109
  - position 138
  - QName 139
  - QName functions, list of 108
  - remove 140
  - replace 140
  - resolve-QName 142
  - reverse 143
  - root 143
  - round 144

- DB2 XQuery functions *(continued)*
  - round-half-to-even 145
  - sequence functions, list of 107
  - sqlquery 3, 146
  - starts-with 147
  - string 148
  - string functions, list of 105
  - string-join 148
  - string-length 149
  - string-to-codepoints 149
  - subsequence 150
  - substring 151
  - substring-after 151
  - substring-before 152
  - sum 153
  - tokenize 154
  - translate 155
  - true 156
  - unordered 157
  - upper-case 157
  - xmlcolumn 2, 158
  - zero-or-one 159
- DB2 XQuery, overview 1
- DB2-defined functions 105
- decimal data type 29
- declarations
  - boundary-space 40
  - construction 41
  - copy-namespaces 41
  - default element/type namespace declarations 42
  - default function namespace 42
  - empty order 43
  - namespace 44
  - ordering mode 44
  - prolog 39
  - version 39
- deep-equal function 118
- default element/type namespace declarations 42
- default function namespace declarations 42
- default-collation function 120
- descendant axis 59
- descendant-or-self axis 59
- direct constructors
  - comment 88
  - description 75
  - element
    - description 76
    - processing instruction 87
    - whitespace in element 80
- distinct-values function 120
- document nodes
  - constructing 85
  - description 7
- document order 10
- double data type 29
- duration data type 30
- duration data types, list of 20
- dynamic context, expressions 47

## E

- effective boolean value 52
- element nodes 8
- elements
  - computed constructors 83
  - direct constructors 76
  - in-scope namespaces 82
- empty function 121
- empty order declarations 43
- empty sequences, ordering 43
- enclosed expressions
  - in constructors 75
- ends-with function 121
- ENTITY data type 31
- entity references 53
- evaluating expressions 47
- exactly-one function 122
- exists function 122
- expanded QNames
  - converting 142
  - description 12
- expressions
  - arithmetic 67
  - atomization 50
  - cast 102
  - combining node sequences 66
  - comparison
    - general 71
    - nodes 73
    - overview 69
    - value 69
  - conditional 100
  - constructing sequences 64
  - constructors
    - computed attribute 84
    - computed comment 88
    - computed element 83
    - computed processing instruction 87
    - description 75
    - direct comment 88
    - direct element 76
    - direct processing instruction 87
    - document node 85
    - in-scope namespaces 82
    - namespace declaration attributes 79
    - processing instruction 86
    - text node 86
  - dynamic context 47
  - effective boolean value 52
  - enclosed in constructors 75
  - filter 65
- FLWOR
  - example 97
  - for and let clauses together 92
  - for and let clauses, comparison 93
  - for and let clauses, overview 90
  - for and let clauses, variable scope 93
  - for clauses 90
  - let clauses 92
  - order by clauses 95
  - overview 89

- expressions (*continued*)
  - FLWOR (*continued*)
    - return clauses 97
    - syntax 89
    - where clauses 94
  - focus 47
  - logical 73
  - order of results 48
  - overview 47
  - path
    - abbreviated syntax 61
    - description 56
    - syntax 57
  - precedence 47
  - predicates 63
  - primary
    - character references 54
    - context item 55
    - entity references 53
    - function calls 55
    - literals 52
    - overview 52
    - parenthesized 55
    - variable references 54
  - processing 47
  - quantified 101
  - range 64
  - sequence 64
  - subtype substitution 50
  - type promotion 51

## F

- false function 123
- filter expressions 65
- float data type 31
- floor function 123
- FLWOR expressions
  - example 97
  - for and let clauses
    - comparison 93
    - in the same expression 92
    - overview 90
    - variable scope 93
  - for clauses 90
  - let clauses
    - description 92
  - order by clauses 95
  - overview 89
  - return clauses 97
  - syntax 89
  - where clauses 94
- focus of expressions 47
- for clauses
  - description 90
- forward axis 59
- function calls 55
- functions
  - DB2 XQuery
    - abs 109
    - avg 110

- functions (*continued*)
  - DB2 XQuery (*continued*)
    - boolean 111
    - boolean functions, list of 106
    - categories 105
    - ceiling 112
    - codepoints-to-string 112
    - compare 113
    - concat 114
    - contains 114
    - count 115
    - current-date 115
    - current-dateTime 116
    - current-time 116
    - data 117
    - date functions, list of 107
    - dateTime 117
    - deep-equal 118
    - default-collation 120
    - distinct-values 120
    - empty 121
    - ends-with 121
    - exactly-one 122
    - exists 122
    - false 123
    - floor 123
    - implicit-timezone 124
    - index-of 125
    - ins-scope-prefixes 124
    - insert-before 126
    - last 127
    - lists of 105
    - local-name 127
    - local-name-from-QName 128
    - lower-case 128
    - matches 129
    - max 130
    - min 131
    - name 132
    - namespace-uri 133
    - namespace-uri-for-prefix 134
    - namespace-uri-from-QName 134
    - node functions, list of 108
    - node-name 135
    - normalize-space 135
    - normalize-unicode 136
    - not 137
    - number 137
    - number functions, list of 106
    - one-or-more 138
    - other functions, list of 109
    - position 138
    - QName 139
    - QName functions, list of 108
    - remove 140
    - replace 140
    - resolve-QName 142
    - reverse 143
    - root 143
    - round 144
    - round-half-to-even 145



- functions (*continued*)
  - DB2 XQuery (*continued*)
    - sequence functions, list of 107
    - sqlquery 146
    - starts-with 147
    - string 148
    - string functions, list of 105
    - string-join 148
    - string-length 149
    - string-to-codepoints 149
    - subsequence 150
    - substring 151
    - substring-after 151
    - substring-before 152
    - sum 153
    - tokenize 154
    - translate 155
    - true 156
    - unordered 157
    - upper-case 157
    - xmlcolumn 158
    - zero-or-one 159

## G

- gDay data type 31
- general comparisons 71
- generic data types, list of 19
- gMonth data type 32
- gMonthDay data type 32
- gYear data type 33
- gYearMonth data type 33

## H

- hexBinary data type 33
- hierarchy, data type 17
- hierarchy, nodes 10

## I

- ID data type 33
- identity of nodes 10
- IDREF data type 33
- if-then-else expressions
  - description 100
- implicit-timezone function 124
- in-scope namespaces 82
- in-scope-prefixes function 124
- index-of function 125
- insert-before function 126
- int data type 34
- integer data type 34
- items in sequences 4

## K

- kind tests 60

## L

- language data type 34
- last function 127
- legal notices 169
- let clauses
  - description 92
- limits
  - size 162
  - XQuery data types 161
- literals 52
- local-name function 127
- local-name-from-QName function 128
- logical expressions 73
- long data type 34
- lower-case function 128

## M

- matches function 129
- max function 130
- min function 131

## N

- Name data type 34
- name function 132
- name tests 59
- namespace declaration attributes 79
- namespace declarations 44
- namespace-uri function 133
- namespace-uri-for-prefix function 134
- namespace-uri-from-QName function 134
- namespaces
  - binding a prefix 80
  - declaring 44
  - default element/type 42, 80
  - function default 42
  - in-scope 82
  - setting default 79
- NCName data type 34
- negativeInteger data type 34
- NMTOKEN data type 35
- node tests 59
- node-name function 135
- nodes
  - attribute 8
  - combining sequences 66
  - comment
    - computed constructors 88
    - constructing, overview 88
    - description 9
    - direct constructors 88
  - comparing 73
  - document
    - constructing 85
    - description 7
  - duplicate 10
  - element 8
  - hierarchy 10
  - identity 10

- nodes (*continued*)
  - overview 5, 7
  - processing instruction
    - constructing 86
    - description 9
  - properties 7
  - string values 10
  - text
    - constructing 86
    - description 9
  - typed values 10
- nonNegativeInteger data type 35
- nonPositiveInteger data type 35
- normalize-space function 135
- normalize-unicode function 136
- normalizedString data type 35
- not function 137
- NOTATION data type 35
- number function 137
- number functions, list of 106
- numeric data types, list of 20
- numeric literal 52
- numeric predicates 63

## O

- one-or-more function 138
- operators
  - precedence 47
- or operator 73
- order by clauses 95
- order of processing 95
- order of results 48
- ordering
  - books 165
- ordering mode declarations 44

## P

- parent axis 59
- parentheses, precedence of operations 47
- parenthesized expressions 55
- path expressions
  - abbreviated and unabbreviated syntax 61
  - axis steps 58
  - description 56
  - syntax 57
- position function 138
- positional predicates 63
- positiveInteger data type 35
- precedence
  - operators and expressions 47
- predicates
  - in expressions 63
- primary expressions 52
- primitive type casting 23
- problem determination 167
- processing instruction nodes
  - constructing 86
  - description 9
- processing order 95

- prologs
  - boundary-space declarations 40
  - construction declarations 41
  - copy-namespace declarations 41
  - default element/type namespace declarations 42
  - default function namespace declarations 42
  - empty order declarations 43
  - namespace declarations 44
  - ordering mode declarations 44
  - syntax 39
  - version declarations 39

## Q

- QName data type 36
- QName function 139
- QNames (qualified names)
  - expanded, converting 142
  - overview 12
- qualified names (QNames)
  - expanded, converting 142
  - overview 12
- qualified expressions 101
- queries
  - structure 1
- query languages
  - case sensitivity 14
  - comments 15
  - XML data 2

## R

- range expressions 64
- remove function 140
- replace function 140
- resolve-QName function 142
- resources
  - XQuery 15
- results
  - order for expressions 48
- return clauses 97
- reverse axis 59
- reverse function 143
- root function 143
- round function 144
- round-half-to-even function 145

## S

- self axis 59
- sequence expressions 64
- sequence functions, list of 107
- sequences
  - atomization 50
  - constructing 64
  - description 4
  - effective boolean value 52
  - empty 43
  - nodes, combining 66
- setters, prolog 39
- short data type 36

- specifications
  - XQuery 15
- sqlquery function 3, 146
- starts-with function 147
- statically known namespaces 82
- string data type 36
- string data types, list of 19
- string function 148
- string functions, list of 105
- string literal 52
- string values of nodes 10
- string-join function 148
- string-length function 149
- string-to-codepoints function 149
- subsequence function 150
- substring function 151
- substring-after function 151
- substring-before function 152
- subtype substitution 50
- sum function 153
- syntax
  - abbreviated 61
  - FLWOR expressions 89

## T

- technical library 163
- text nodes
  - constructing 86
  - description 9
- time data type 36
- time data types, list of 20
- timezone, implicit 124
- token data type 37
- tokenize function 154
- translate function 155
- troubleshooting
  - information 167
- true function 156
- type casting 23
- type hierarchy 17
- type promotion 51
- typed values of nodes 10
- types
  - see data types 19

## U

- Unicode characters 54
- unordered function 157
- unsignedByte data type 37
- unsignedInt data type 37
- unsignedLong data type 37
- unsignedShort data type 37
- untyped data type 37
- untyped data types, list of 19
- untypedAtomic data type 37
- upper-case function 157
- URI
  - binding a namespace prefix 80

## V

- value comparisons 69
- values, atomic 5
- variables
  - in scope in for and let clauses 93
  - positional in for clauses 91
  - references 54
- version declarations 39

## W

- where clauses
  - description 94
- whitespace
  - boundary 40
  - description 14
  - in direct element constructors 80

## X

- XML data
  - querying in DB2 database 2
- xmlcolumn function 2, 158
- XMLEXISTS function 2
- XMLQUERY function 2
- XMLTABLE function 2
- XQuery
  - invoking from SQL 2
  - overview 1
  - resources 15
- XQuery and XPath data model 4
- XQuery-defined functions 105

## Y

- yearMonthDuration data type 38

## Z

- zero-or-one function 159







Printed in USA

SC18-9796-00

