IBM DB2 10.1
for Linux, UNIX, and Windows

XQuery Reference

IBM

IBM DB2 10.1
for Linux, UNIX, and Windows

XQuery Reference

IBM

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 http://www.ibm.com/shop/publications/order
- To find your local IBM representative, go to the IBM Directory of Worldwide Contacts at 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.

Contents

iii

About this book

The XQuery Reference describes the XQuery language used by a DB2® database to work with XML data.

It includes information about XQuery concepts, data types, language elements, XQuery-defined functions, and DB2-defined functions. The reference also includes information about DB2 XQuery size limits and limits for XQuery data types.

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 and modifying 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 use the XQuery language to 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.
- Update existing XML data.

Components of an XQuery query

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 on page 2 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.

```
  xquery version "1.0";
  declare default element namespace "http://posample.org";
```

Prolog

```
<price_list>{for $prod in db2-fn:xmlcolumn("PRODUCT.DESCRIPTION")/product/description
      order by xs:decimal($prod/price) descending
      return <product>{$prod/name, $prod/price}</product>}
</price_list>
```

Query body

Figure 1. Structure of a typical query in XQuery

Comparison of XQuery to SQL

DB2 databases support 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).

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. The function supports using parameters to pass values to the SQL statement.

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.

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.

Sequences and items

An instance of the XQuery and XPath data model (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 entry in the following list is a sequence:

- 36
-
- (2, 3, 4)
- (36, , "cat")
- ()

In addition the entries in the list, an XML document stored in an XML column in a DB2 database is a sequence.

The 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.
- An 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.

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.

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 pid="10">
    <description>
      <name>Fleece jacket</name>
      <price>19.99</price>
    </description>
  </product>
  <product pid="11">
    <description>
      <name>Nylon pants</name>
      <price>9.99</price>
    </description>
  </product>
</products>
```

Figure 2 on page 7 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).
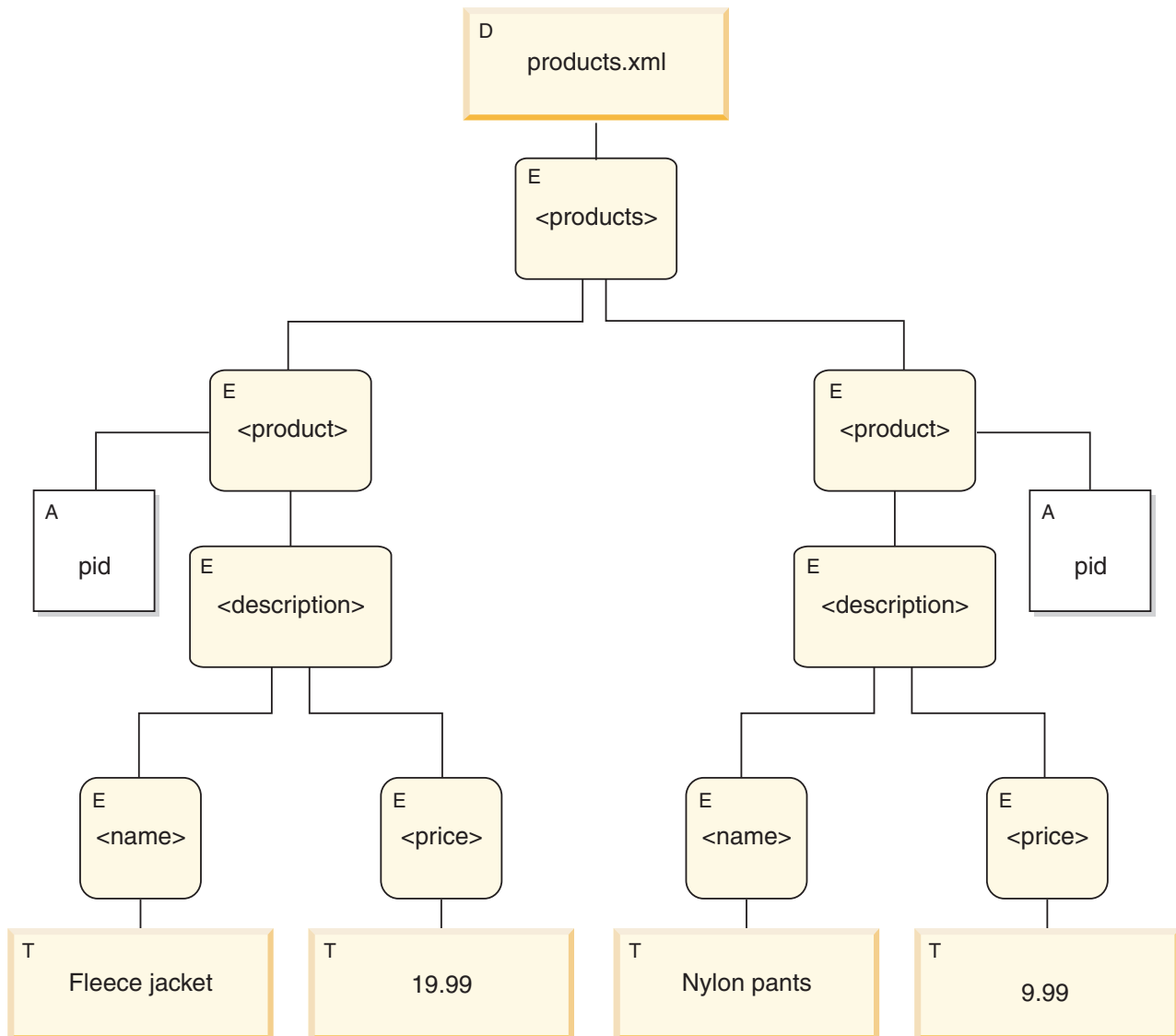
Figure 2. Data model diagram for products.xml document

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`.

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.

Node kinds

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

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.

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.

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.

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.

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.

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.

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.

Nodes in a hierarchy 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.

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.

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 xs:string data type that is the concatenated contents of all its descendant text nodes, in document order.	An instance of the xdt:untypedAtomic data type that is the concatenated contents of all its descendant text nodes, in document order.
Element in an XML 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 an XML 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.

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.

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*).

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*.

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 XQuery 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
xdt	`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.

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.

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.

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 :) might be confusing, but is 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.

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 Recommendation, 23 January 2007. See www.w3.org/TR/2007/REC-xquery-20070123/.
- **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 Recommendation, 23 January 2007. See www.w3.org/TR/2007/REC-xpath-functions-20070123/.
- **XQuery 1.0 and XPath 2.0 Data Model**

  World Wide Web Consortium. *XQuery 1.0 and XPath 2.0 Data Model*. W3C Recommendation, 23 January 2007. See www.w3.org/TR/2007/REC-xpath-datamodel-20070123/.
- **XML Query Use Cases**

  World Wide Web Consortium. *XML Query Use Cases*. W3C Working Group Note, 23 March 2007. See www.w3.org/TR/2007/NOTE-xquery-use-cases-20070323/.
- **XML Schema**

  World Wide Web Consortium. *XML Schema, Parts 0, 1, and 2*. W3C Recommendation, 28 October 2004. See www.w3.org/TR/2004/REC-xmlschema-0-20041028/, www.w3.org/TR/2004/REC-xmlschema-1-20041028/, and www.w3.org/TR/2004/REC-xmlschema-2-20041028/.
- **XML Names**

  World Wide Web Consortium. *Namespaces in XML 1.0 (Second Edition)*. W3C Recommendation, 16 August 2006. See www.w3.org/TR/2006/REC-xml-names-20060816/.
- **Updating XML**

  World Wide Web Consortium. *XQuery Update Facility*. W3C Working Draft, 11 July 2006. See www.w3.org/TR/2006/WD-xqupdate-20060711/.

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.

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.
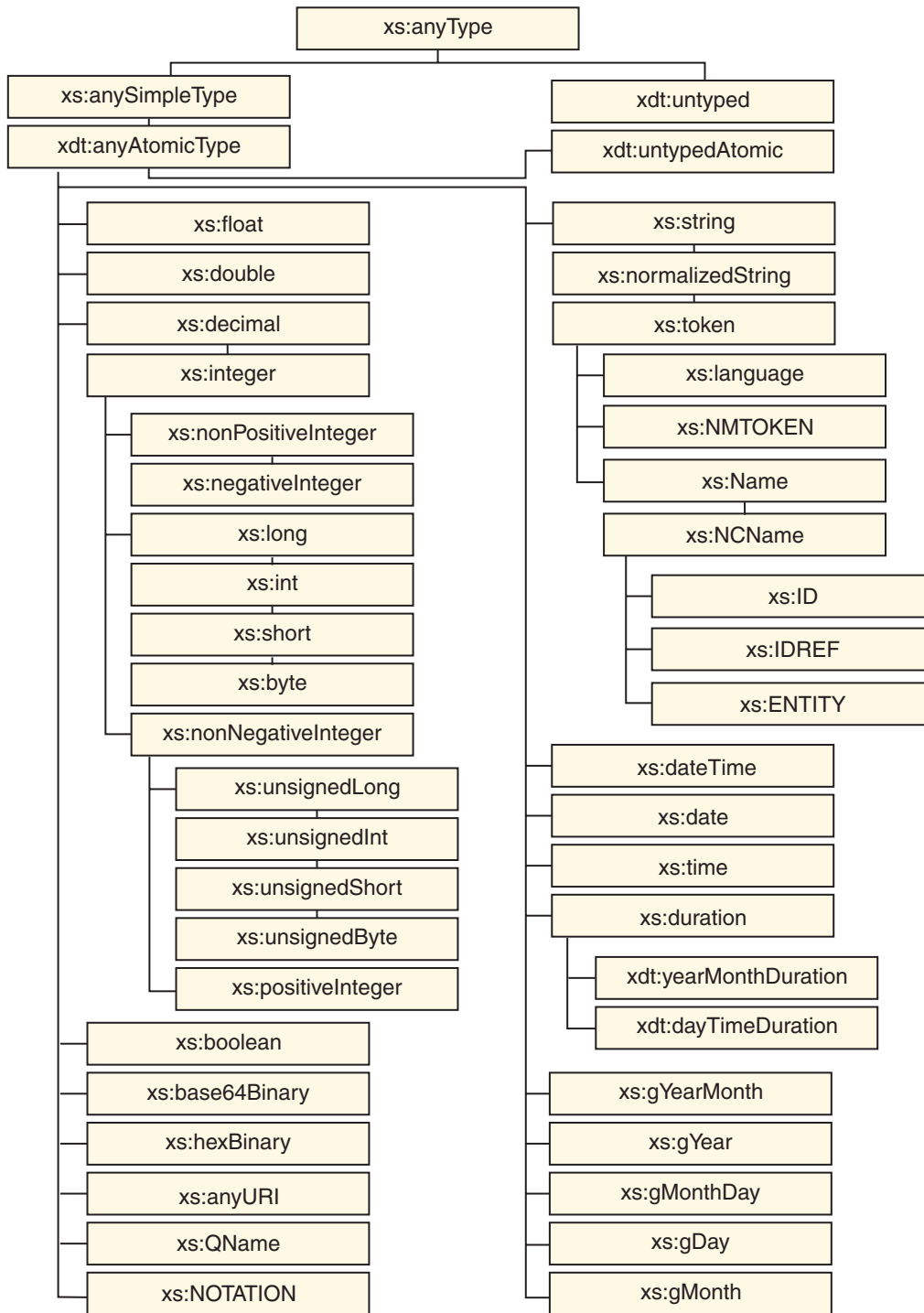
Figure 3. DB2 XQuery type hierarchy

Types by category

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

Generic data types

Table 3. Generic data types

Type	Description
"anyType data type" on page 27	The xs:anyType data type encompasses any sequence of zero or more nodes and zero or more atomic values.
"anySimpleType data type" on page 27	The xs:anySimpleType data type 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 can be any sequence of atomic values. Derived from the xs:anyType data type.
"anyAtomicType data type" on page 27	The xdt:anyAtomicType data type 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. Derived from the xs:anySimpleType data type.

Untyped data types

Table 4. Untyped data types

Type	Description
"untyped data type" on page 41	The xdt:untyped data type denotes a node that has not been validated by an XML schema. Derived from data type xs:anyType.
"untypedAtomic data type" on page 41	The xdt:untypedAtomic data type denotes an atomic value that has not been validated by an XML schema. Derived from data type xdt:anyAtomicType.

String data types

Table 5. String data types

Type	Description
"string data type" on page 39	The xs:string data type represents a character string. Derived from data type xdt:anyAtomicType.
"normalizedString data type" on page 38	The xs:normalizedString data type represents a white space-normalized string. Derived from data type xs:string.
"token data type" on page 40	The xs:token data type represents a tokenized string. Derived from the xs:normalizedString data type.
"language data type" on page 36	The xs:language data type represents a natural language identifier as defined by *RFC 3066*. Derived from data type xs:token.
"NMTOKEN data type" on page 37	The xs:NMTOKEN data type represents the NMTOKEN attribute type from *XML 1.0 (Third Edition)*. Derived from the xs:token data type.

Table 5. String data types (continued)

Type	Description
"Name data type" on page 37	The xs:Name data type represents an XML Name. Derived from the xs:token data type.
"NCName data type" on page 37	The xs:NCName data type represents an XML noncolonized name. Derived from the xs:Name data type.
"ID data type" on page 36	The xs:ID data type represents the ID attribute type from *XML 1.0 (Third Edition)*. Derived from xs:NCName data type.
"IDREF data type" on page 36	The xs:IDREF data type represents the IDREF attribute type from *XML 1.0 (Third Edition)*. Derived from the xs:NCName data type.
"ENTITY data type" on page 33	The xs:ENTITY data type represents the ENTITY attribute type from *XML 1.0 (Third Edition)*. Derived from the xs:NCName data type.

Numeric data types

Table 6. Numeric data types

Type	Description
"decimal data type" on page 31	The xs:decimal data type represents a subset of the real numbers that can be represented by decimal numerals. Derived from data type xdt:anyAtomicType.
"float data type" on page 33	The xs:float data type is patterned after the IEEE single-precision 32-bit floating point type. Derived from data type xdt:anyAtomicType.
"double data type" on page 31	The xs:double data type is patterned after the IEEE double-precision 64-bit floating point type. Derived from data type xdt:anyAtomicType.
"int data type" on page 36	The xs:int data type represents an integer that is less than or equal to 2 147 483 647 and greater than or equal to -2 147 483 648. Derived from the xs:long data type.
"nonPositiveInteger data type" on page 38	The xs:nonPositiveInteger data type represents an integer that is less than or equal to zero. Derived from the xs:integer data type.
"negativeInteger data type" on page 37	The xs:negativeInteger data type represents an integer that is less than zero. Derived from data type xs:nonPositiveInteger.
"nonNegativeInteger data type" on page 37	The xs:nonNegativeInteger data type represents an integer that is greater than or equal to zero. Derived from the xs:integer data type.

Table 6. Numeric data types (continued)

Type	Description
"long data type" on page 37	The xs:long data type 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. Derived from data type xs:integer.
"integer data type" on page 36	The xs:integer data type 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. Derived from xs:decimal data type.
"short data type" on page 39	The xs:short data type represents an integer that is less than or equal to 32 767 and greater than or equal to -32 768. Derived from the xs:int data type.
"byte data type" on page 28	The xs:byte data type represents an integer that is less than or equal to 127 and greater than or equal to -128. Derived from the xs:short data type.
"unsignedLong data type" on page 40	The xs:unsignedLong data type represents an unsigned integer that is less than or equal to 9 223 372 036 854 775 807. Derived from the xs:nonNegativeInteger data type.
"unsignedInt data type" on page 40	The xs:unsignedInt data type represents an unsigned integer that is less than or equal to 4 294 967 295. Derived from xs:unsignedLong data type.
"unsignedShort data type" on page 40	The xs:unsignedShort data type represents an unsigned integer that is less than or equal to 65 535. Derived from the xs:unsignedInt data type.
"unsignedByte data type" on page 40	The xs:unsignedByte data type represents an unsigned integer that is less than or equal to 255. Derived from xs:unsignedShort data type.
"positiveInteger data type" on page 38	The xs:positiveInteger data type represents a positive integer that is greater than or equal to 1. Derived from the xs:nonNegativeInteger data type.

Date, time, and duration data types

Table 7. Date, time, and duration data types

Type	Description
"duration data type" on page 32	The xs:duration data type represents a duration of time that is expressed by the Gregorian year, month, day, hour, minute, and second components. Derived from data type xdt:anyAtomicType.

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

Type	Description
"yearMonthDuration data type" on page 41	The xdt:yearMonthDuration data type represents a duration of time that is expressed by the Gregorian year and month components. Derived from the xs:duration data type.
"dayTimeDuration data type" on page 30	The xdt:dayTimeDuration data type represents a duration of time that is expressed by days, hours, minutes, and seconds components. Derived from the xs:duration data type.
"dateTime data type" on page 28	The xs:dateTime data type 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. Derived from data type xdt:anyAtomicType.
"date data type" on page 28	The xs:date data type represents an interval of exactly one day in duration that begins on the first moment of a specific day. The xs:date data type consists of year, month, and day properties that are expressed as integer values and an optional time zone indicator. Derived from data type xdt:anyAtomicType.
"time data type" on page 39	The xs:time data type represents an instant of time that recurs every day. Derived from data type xdt:anyAtomicType.
"gYearMonth data type" on page 35	The xs:gYearMonth data type represents a specific Gregorian month in a specific Gregorian year. Gregorian calendar months are defined in *ISO 8601*. Derived from data type xdt:anyAtomicType.
"gYear data type" on page 35	The xs:gYear data type represents a Gregorian calendar year. Gregorian calendar years are defined in *ISO 8601*. Derived from data type xdt:anyAtomicType.
"gMonthDay data type" on page 34	The xs:gMonthDay data type represents a Gregorian date that recurs. Gregorian calendar dates are defined in *ISO 8601*. Derived from data type xdt:anyAtomicType.
"gDay data type" on page 34	The xs:gDay data type represents a Gregorian day that recurs. Gregorian calendar days are defined in *ISO 8601*. Derived from data type xdt:anyAtomicType.
"gMonth data type" on page 34	The xs:gMonth data type represents a Gregorian month that recurs every year. Gregorian calendar months are defined in *ISO 8601*. Derived from data type xdt:anyAtomicType.

Other data types

Table 8. Other data types

Type	Description
"boolean data type" on page 27	The xs:boolean data type supports the mathematical concept of binary-valued logic: true or false. Derived from data type xdt:anyAtomicType.
"anyURI data type" on page 27	The xs:anyURI data type represents a Uniform Resource Identifier (URI). Derived from data type xdt:anyAtomicType.
"QName data type" on page 38	The xs:QName data type 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. Derived from data type xdt:anyAtomicType.
"NOTATION data type" on page 38	The xs:NOTATION data type represents the NOTATION attribute type from *XML 1.0 (Third Edition)*. Derived from data type xdt:anyAtomicType.
"hexBinary data type" on page 36	The xs:hexBinary data type represents hex-encoded binary data. Derived from data type xdt:anyAtomicType.
"base64Binary data type" on page 27	The xs:base64Binary data type represents base64-encoded binary data. Derived from data type xdt:anyAtomicType.

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."

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.

When casting a value to a data type, you can use the castable expression to test whether the value can be cast to the data type.

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 returned.

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

Source data type	Target uA	Target string	Target float	Target double	Target decimal	Target integer	Target dur	Target yMD	Target dTD	Target 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
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 time	Target date	Target gYM	Target gYr	Target gMD	Target gDay	Target gMon	Target bool	Target b64	Target hxB	Target aURI	Target QN	Target 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

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

Source data type	Target time	Target date	Target gYM	Target gYr	Target gMD	Target gDay	Target gMon	Target bool	Target b64	Target hxB	Target aURI	Target QN	Target NOT
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

anyAtomicType data type

The xdt:anyAtomicType data type 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. Derived from the xs:anySimpleType data type.

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

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

anySimpleType data type

The xs:anySimpleType data type 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 can be any sequence of atomic values. Derived from the xs:anyType data type.

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

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

anyType data type

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

anyURI data type

The xs:anyURI data type represents a Uniform Resource Identifier (URI). Derived from data type xdt:anyAtomicType.

The lexical form of The xs:anyURI data type 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 xs:base64Binary data type represents base64-encoded binary data. Derived from data type xdt:anyAtomicType.

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 xs:boolean data type supports the mathematical concept of binary-valued logic: true or false. Derived from data type xdt:anyAtomicType.

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

byte data type

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

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 xs:date data type represents an interval of exactly one day in duration that begins on the first moment of a specific day. The xs:date data type consists of year, month, and day properties that are expressed as integer values and an optional time zone indicator. Derived from data type xdt:anyAtomicType.

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-dd*zzzzzz. 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 29 for more information about the format for this property.

dateTime data type

The xs:dateTime data type 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. Derived from data type xdt:anyAtomicType.

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.ssssszzzzzz*. 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.

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

zzzzzz
: Optional. If present, represents the timezone. See "Timezone indicator" 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 xdt:dayTimeDuration data type represents a duration of time that is expressed by days, hours, minutes, and seconds components. Derived from the xs:duration data type.

The range that can be represented by this data type is from -P83333333333333Y3M11574074074DT1H46M39.999999S to P83333333333333Y3M11574074074DT1H46M39.999999S (or -999999999999999 months and -999999999999999.999999 seconds to 999999999999999 months and 999999999999999.999999 seconds).

The lexical form of xdt:dayTimeDuration is P*n*DT*n*H*n*M*n*S, which is a reduced form of the *ISO 8601* format. The following abbreviations describe this form:

P The duration designator.

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

T The date and time separator.

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

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

*n***S** *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
P4DT251M
```

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

DB2 database system stores xdt:dayTimeDuration values in a normalized form. In the normalized form, the seconds and minutes components are less than 60, and the hours component is less than 24. Each multiple of 60 seconds is converted to one minute, each multiple of 60 minutes to one hour, and each multiple of 24 hours to one day. For example, the following XQuery expression invokes a constructor function specifying a dayTimeDuration of 63 days, 55 hours, and 81 seconds:

```
xquery
xdt:dayTimeDuration("P63DT55H81S")
```

In the duration, 55 hours is converted to 2 days and 7 hours, and 81 seconds is converted to 1 minute and 21 seconds. The expression returns the normalized dayTimeDuration value P65DT7H1M21S.

decimal data type

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

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 zeros are optional. If the fractional part is zero, the period and any following zeros can be omitted. The following numbers are valid examples of this data type:

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

double data type

The xs:double data type is patterned after the IEEE double-precision 64-bit floating point type. Derived from data type xdt:anyAtomicType.

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:double("INF")`.

duration data type

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

The range that can be represented by this data type is from -P83333333333333Y3M11574074074DT1H46M39.999999S to P83333333333333Y3M11574074074DT1H46M39.999999S (or -999999999999999 months and -999999999999999.999999 seconds to 999999999999999 months and 999999999999999.999999 seconds).

The lexical form of xs:duration is the *ISO 8601* extended format P*n*Y*n*M*n*DT*n*H*n*M*n*S. The following abbreviations describe the extended format:

P   The duration designator.

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

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

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

T   The date and time separator.

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

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

*n*S   *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.

The DB2 database system stores xs:duration values in a normalized form. In the
normalized form, the seconds and minutes components are less than 60, the hours
component is less than 24, and the months component is less than 12. Each
multiple of 60 seconds is converted to one minute, each multiple of 60 minutes to
one hour, each multiple of 24 hours to one day, and each multiple of 12 months to
one year. For example, the following XQuery expression invokes a constructor
function specifying a duration of 2 months, 63 days, 55 hours, and 91 minutes:

```
xquery
xs:duration("P2M63DT55H91M")
```

In the duration, 55 hours is converted to 2 days and 7 hours, and 91 minutes is
converted to 1 hour and 31 minutes. The expression returns the normalized
duration value P2M65DT8H31M.

ENTITY data type

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

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

float data type

The xs:float data type is patterned after the IEEE single-precision 32-bit floating
point type. Derived from data type xdt:anyAtomicType.

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 xs:gDay data type represents a Gregorian day that recurs. Gregorian calendar days are defined in *ISO 8601*. Derived from data type xdt:anyAtomicType.

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 `---ddzzzzzz`, 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 29 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 xs:gMonth data type represents a Gregorian month that recurs every year. Gregorian calendar months are defined in *ISO 8601*. Derived from data type xdt:anyAtomicType.

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 29 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 xs:gMonthDay data type represents a Gregorian date that recurs. Gregorian calendar dates are defined in *ISO 8601*. Derived from data type xdt:anyAtomicType.

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 29 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 xs:gYear data type represents a Gregorian calendar year. Gregorian calendar years are defined in *ISO 8601*. Derived from data type xdt:anyAtomicType.

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 29 for more information about the format for this property.

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

gYearMonth data type

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

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 29 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 xs:hexBinary data type represents hex-encoded binary data. Derived from data type xdt:anyAtomicType.

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

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

IDREF data type

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

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

int data type

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

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 xs:integer data type 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. Derived from xs:decimal data type.

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 xs:language data type represents a natural language identifier as defined by *RFC 3066*. Derived from data type xs:token.

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 xs:long data type 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. Derived from data type xs:integer.

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 xs:Name data type represents an XML Name. Derived from the xs:token data type.

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

NCName data type

The xs:NCName data type represents an XML noncolonized name. Derived from the xs:Name data type.

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

negativeInteger data type

The xs:negativeInteger data type represents an integer that is less than zero. Derived from data type xs:nonPositiveInteger.

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

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

nonNegativeInteger data type

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

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 can 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 xs:nonPositiveInteger data type represents an integer that is less than or equal to zero. Derived from the xs:integer data type.

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 can be negative (-) or can 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 xs:normalizedString data type represents a white space-normalized string. Derived from data type xs: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 xs:NOTATION data type represents the NOTATION attribute type from *XML 1.0 (Third Edition)*. Derived from data type xdt:anyAtomicType.

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

positiveInteger data type

The xs:positiveInteger data type represents a positive integer that is greater than or equal to 1. Derived from the xs:nonNegativeInteger data type.

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 xs:QName data type 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. Derived from data type xdt:anyAtomicType.

The lexical form of The xs:QName data type 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 xs:short data type represents an integer that is less than or equal to 32 767 and greater than or equal to -32 768. Derived from the xs:int data type.

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 xs:string data type represents a character string. Derived from data type xdt:anyAtomicType.

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 xs:time data type represents an instant of time that recurs every day. Derived from data type xdt:anyAtomicType.

The lexical form of xs:time is *hh:mm:ss.ssssssszzzzzz*. 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.

.ssssss
> 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 29 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 xs:token data type represents a tokenized string. Derived from the xs:normalizedString data type.

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 xs:unsignedByte data type represents an unsigned integer that is less than or equal to 255. Derived from xs:unsignedShort data type.

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

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

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 xs:unsignedShort data type represents an unsigned integer that is less than or equal to 65 535. Derived from the xs:unsignedInt data type.

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 xdt:untyped data type denotes a node that has not been validated by an XML schema. Derived from data type xs:anyType.

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 xdt:untypedAtomic data type denotes an atomic value that has not been validated by an XML schema. Derived from data type xdt:anyAtomicType.

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

yearMonthDuration data type

The xdt:yearMonthDuration data type represents a duration of time that is expressed by the Gregorian year and month components. Derived from the xs:duration data type.

The range that can be represented by this data type is from -P83333333333333Y3M to P83333333333333Y3M (or -999999999999999 to 999999999999999 months).

The lexical form of xdt:yearMonthDuration is P*n*Y*n*M, which is a reduced form of the *ISO 8601* format. The following abbreviations describe this form:

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

*n*M   *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.

The DB2 database system stores xdt:yearMonthDuration values in a normalized form. In the normalized form, the months component is less than 12. Each multiple of 12 months is converted to one year. For example, the following XQuery expression invokes a constructor function specifying a yearMonthDuration of 20 years and 30 months:

```
xquery
xdt:yearMonthDuration("P20Y30M")
```

In the duration, 30 months is converted to 2 years and 6 months. The expression returns the normalized yearMonthDuration value P22Y6M.

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 XQuery 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 XQuery also supports the following setters. However, they do not change the processing environment because DB2 XQuery 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
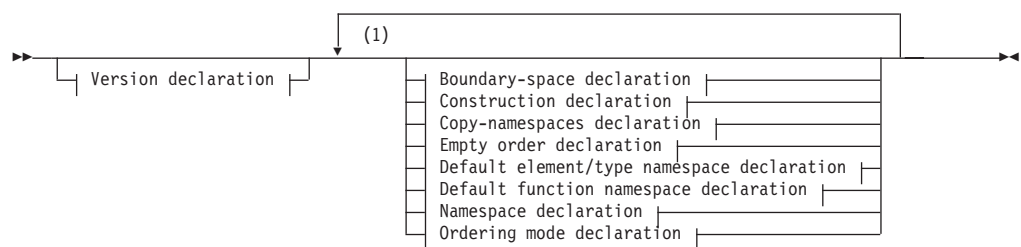
Notes:

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

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 XQuery.

If present, the version declaration must be at the beginning of the prolog. The only version that is supported by DB2 XQuery 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 XQuery 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;
```

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 **strip**. For DB2 XQuery, when the construction mode is **strip**, the type of a constructed element node is xdt:untypedAtomic; all element nodes copied during node construction receive the type xdt:untypedAtomic, and all attribute nodes copied during node construction receive the type xdt:untypedAtomic.

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

Syntax

```
►►──declare──construction────strip────;─────────────────────────────►◄
```

strip
> For DB2 XQuery, specifies the type of a constructed element node is xdt:untypedAtomic; all element nodes copied during node construction receive the type xdt:untypedAtomic, and all attribute nodes copied during node construction receive the type xdt:untypedAtomic.

Example

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

```
declare construction strip;
```

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 XQuery, 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;
```

Default element/type namespace declaration

The default element/type namespace declaration in the query prolog specifies the namespace to use for the unprefixed 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 unprefixed element and type names are not in any namespace.

The default element/type namespace does not apply to unqualified attribute names. Unprefixed 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 unprefixed 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.

Default function namespace declaration

The default function namespace declaration in the query prolog specifies a namespace URI that is used for unprefixed 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 XQuery returns an error if the local name for an unprefixed 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()`.

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;
```

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 XQuery 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 XQuery, see "Order of results in XQuery expressions" on page 52.

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 XQuery, see "Order of results in XQuery expressions" on page 52.

Example

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

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	`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
xdt	`http://www.w3.org/2005/xpath-datatypes`	XQuery type namespace
db2-fn	`http://www.ibm.com/xmlns/prod/db2/functions`	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`.

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.

Expression evaluation and processing

A number of operations are often included in the processing of expressions. These operations include 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.

In DB2 XQuery, updating expressions can be used only within the `modify` clause of a transform expression. For information about the XQuery transform expression, and updating expression processing, see "Transform expression" on page 114 and "Use of updating expressions in a transform expression" on page 111.

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 XQuery 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.

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

Table 12. Precedence in DB2 XQuery (continued)

Operator or expression	Associativity
FLWOR, some, every, if	left-to-right
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
castable	left to right
cast	left-to-right
-(unary), +(unary)	right-to-left
?	left-to-right
/, //	left-to-right
[ ], ( ), {}	left-to-right

Order of results in XQuery expressions

When using DB2 XQuery, some kinds of XQuery expressions return sequences in a deterministic order while other kinds of expressions return sequences in a non-deterministic order.

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
```

- Updating expressions that add elements and use explicit position keywords return results with the added elements in the position specified. For example, the following updating expression uses the **as first into** keywords and inserts the element <status>current</status> as the first item element in the customerinfo element:

```
xquery
transform
copy $mycust := db2-fn:sqlquery('select info from customer where cid = 1001')
modify
   do insert <status>current</status> as first into $mycust/customerinfo
return $mycust
```

- 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 **order by** clause	Determined by the **order by** clause	The following expression returns a sequence of product elements in ascending order by price: `for $p in /product order by $p/price return $p`
Updating expressions	Use of keywords that specify a position when adding an element	Determined by the updating expression keywords	When inserting an element using the keywords **as last into**, the element is added as the last child of the specified node.
Expressions with **union**, **intersect**, or **except** operators	None	Document order	`$managers union $students`

Table 13. Summary of ordering of results in XQuery expressions (continued)

Expression kind	Conditions for a deterministic ordering	Ordering of results	Example

Path expressions	• 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.
		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 **to** operator	None	Sequence of integers in ascending order	`15 to 25`
Expressions that contain comma operators	All the operands are sequences with deterministic order	Return results are in the order of their operands	`(5, 10, 15 to 20,` `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.

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

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.

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.

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

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

Description of value	EBV returned
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 if 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**)

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 &amp; Jerry&apos;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."

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
<	<
>	>
&	&
"	"
'	'

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 € or € represents the Euro symbol (€).

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 XQuery 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.

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.

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.

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.

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
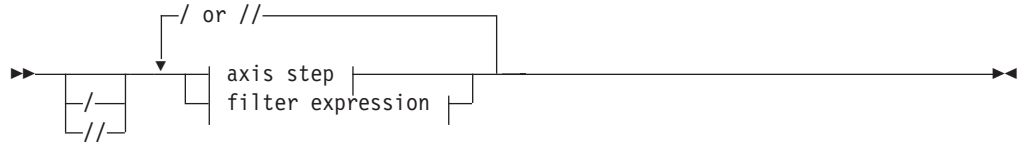
form the sequence of context nodes for the step that follows. Duplicate nodes are eliminated from this combined sequence, based on node identity.

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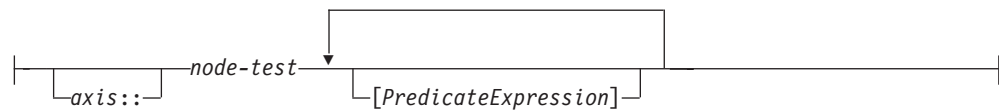
The node sequence that results from a path expression is not guaranteed to be in a specific order. To understand when a path expression returns ordered results, see the topic that describes the order of results in XQuery expressions.
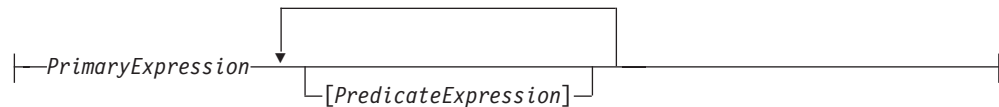
Syntax of path expressions

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.

```
                    ┌── / or // ──────────────┐
                    │▼                         │
►►─┬────┬───────────┬─┤ axis step ├────────────┴──────────────────────────►◄
   ├─/──┤           └─┤ filter expression ├─┘
   └─//─┘
```

axis step:

```
                           ┌──────────────────────┐
                           │▼                      │
├──┬──────────┬──node-test─┴──┬────────────────────────┬──┴──────────────────┤
   └─axis::───┘               └─[PredicateExpression]─┘
```

filter expression:

```
                      ┌──────────────────────┐
                      │▼                      │
├──PrimaryExpression──┴──┬────────────────────────┬──┴──────────────────────┤
                         └─[PredicateExpression]─┘
```

/   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]
```

Axis steps

Axis steps consist of three parts: an optional *axis* to specify a direction of movement; a *node test* to specify the criteria that is used to select nodes; and zero or more *predicates* to 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.

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	Comments
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.

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
QName	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 `child::para`, the name test `para` selects all of the `para` elements on the child 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, `child::*`, the name test * matches all of the elements on the child axis.
NCName:*	Specifies an *NCName* 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 `child::ns1:*`, the name test `ns1:*` matches all of the elements on the child axis that are associated with the namespace that is bound to the prefix `ns1`.
*:*NCName*	Specifies an *NCName* 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 *NCName*. If the axis is an attribute axis, this test matches attribute nodes. On all other axes, this test matches element nodes.	In the expression `child::*:customerinfo`, the name test `*:customerinfo` matches all of the elements on the child axis that have the local name `customerinfo`, 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
node()	Matches any node on the specified axis.	In the expression `child::node()`, the kind test `node()` selects any nodes on the child axis.
text()	Matches any text node on the specified axis.	In the expression `child::text()`, the kind test `text()` selects any text nodes on the child axis.
comment()	Matches any comment node on the specified axis.	In the expression `child::comment()`, the kind test `comment()` selects any comment nodes on the child 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 `child::processing-instruction()`, the kind test `processing-instruction()` selects any processing instruction nodes on the child axis.
element() or element(*)	Matches any element node on the specified axis.	In the expression `child::element()`, the kind test `element()` selects any element nodes on the child axis. In the expression `child::element(*)`, the kind test `element(*)` selects any element nodes on the child axis.
attribute() or attribute(*)	Matches any attribute node on the specified axis.	In the expression `child::attribute()`, the kind test `attribute()` selects any attribute nodes on the child axis. In the expression `child::attribute(*)`, the kind test `attribute(*)` selects any attribute nodes on the child axis.
document-node()	Matches any document node on the specified axis.	In the expression `self::document-node()`, the kind test `document-node()` selects a document node that is the context node.

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 `child::` except when the axis step specifies `attribute()` for the node test. When the axis step specifies an attribute test, an omitted axis is shorthand for `attribute::`.	The path expression `section/para` is an abbreviation for `child::section/child::para`. The path expression `section/attribute()` is an abbreviation for `child::section/attribute::attribute()`.
@	Shorthand abbreviation for `attribute::` .	The path expression `section/@id` is an abbreviation for `child::section/attribute::id`.

Table 19. Abbreviated syntax for path expressions (continued)

Abbreviated syntax	Description	Examples
`//`	Shorthand abbreviation for `/descendant-or-self::node()/`, 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 `div1//para` is an abbreviation for `child::div1/descendant-or-self::node()/child::para` .
`..`	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.

Table 20. Unabbreviated syntax and abbreviated syntax (continued)

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

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 />`.

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).

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)
```

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
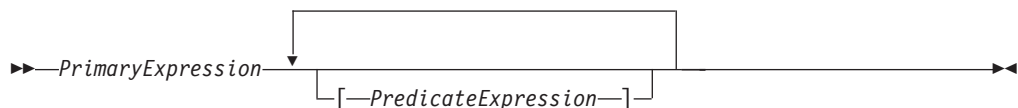
```
►►─ PrimaryExpression ─┬─────────────────────────────┬─►◄
                       └─[─ PredicateExpression ─]───┘
```

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()]
  ```

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
union 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 **union** keyword and the | character are equivalent.
intersect	Takes two node sequences as operands and returns a sequence that contains all of the nodes that occur in both operands.
except	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.

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 (xdt: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 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
A + B	xs:date	xdt:yearMonthDuration	xs:date
A + B	xdt:yearMonthDuration	xs:date	xs:date
A + B	xs:date	xdt:dayTimeDuration	xs:date
A + B	xdt:dayTimeDuration	xs:date	xs:date
A + B	xs:time	xdt:dayTimeDuration	xs:time
A + B	xdt:dayTimeDuration	xs:time	xs:time
A + B	xs:dateTime	xdt:yearMonthDuration	xs:dateTime
A + B	xdt:yearMonthDuration	xs:dateTime	xs:dateTime
A + B	xs:dateTime	xdt:dayTimeDuration	xs:dateTime
A + B	xdt:dayTimeDuration	xs:dateTime	xs:dateTime
A + B	xdt:yearMonthDuration	xdt:yearMonthDuration	xdt:yearMonthDuration
A + B	xdt:dayTimeDuration	xdt:dayTimeDuration	xdt:dayTimeDuration
A - B	numeric	numeric	numeric
A - B	xs:date	xs:date	xdt:dayTimeDuration
A - B	xs:date	xdt:yearMonthDuration	xs:date
A - B	xs:date	xdt:dayTimeDuration	xs:date
A - B	xs:time	xs:time	xdt:dayTimeDuration
A - B	xs:time	xdt:dayTimeDuration	xs:time
A - B	xs:dateTime	xs:dateTime	xdt:dayTimeDuration
A - B	xs:dateTime	xdt:yearMonthDuration	xs:dateTime
A - B	xs:dateTime	xdt:dayTimeDuration	xs:dateTime
A - B	xdt:yearMonthDuration	xdt:yearMonthDuration	xdt:yearMonthDuration
A - B	xdt:dayTimeDuration	xdt:dayTimeDuration	xdt:dayTimeDuration
A * B	numeric	numeric	numeric
A * B	xdt:yearMonthDuration	numeric	xdt:yearMonthDuration
A * B	numeric	xdt:yearMonthDuration	xdt:yearMonthDuration
A * B	xdt:dayTimeDuration	numeric	xdt:dayTimeDuration
A * B	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

Table 23. Valid types for operands of arithmetic expressions (continued)

Operator with operands	Type of operand A	Type of operand B	Result type
A div B	xdt:yearMonthDuration	numeric	xdt:yearMonthDuration
A div B	xdt:dayTimeDuration	numeric	xdt:dayTimeDuration
A div B	xdt:yearMonthDuration	xdt:yearMonthDuration	xs:decimal
A div B	xdt:dayTimeDuration	xdt:dayTimeDuration	xs:decimal
A mod B	numeric	numeric	numeric

Examples

- In the following example, the first expression 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
  ```

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>
  ```

General comparisons

A general comparison compares 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 are =, !=, <, <=, >, and >=.

The following table describes these operators.

Table 25. Value comparison operators in XQuery

Operator	Purpose
=	Returns true if some value in the first sequence is equal to some value in the second sequence.
!=	Returns true if some value in the first sequence is not equal to some value in the second sequence.
<	Returns true if some value in the first sequence is less than some value in the second sequence.
<=	Returns true if some value in the first sequence is less than or equal to some value in the second sequence.

Table 25. Value comparison operators in XQuery (continued)

Operator	Purpose
>	Returns true if some value in the first sequence is greater than some value in the second sequence.
>=	Returns true if some value in the first sequence is greater than or equal to some value in the second sequence.

As illustrated in the Examples section, later, general comparisons are not transitive and note that the = and != operators are not inverses of each other.

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). When the individual atomic values are compared, the following rules are applied to the implicit cast that takes place:

Atomic value in one sequence	Atomic value in other sequence	Type to which untyped value is cast
xdt:untypedAtomic	A numeric type	xs:double If you are working with very large integers, it is possible that precision might be lost. For example, when the 19 digit number −9223372036854775672 is cast to xs:double, the result is −9.223372036854776E18 (three digits of precision are lost). You can avoid this loss of precision by casting the value to an xs:decimal or xs:long type.
xdt:untypedAtomic	xdt:untypedAtomic or xs:string	xs:string
xdt:untypedAtomic	A value other than a numeric type, xdt:untypedAtomic, or xs:string	The type of the other value

If the types are successfully cast, 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. For example, the comparison (1, 2) = (2, 3) returns true because 2 eq 2 is true. If the implicit cast operation fails, the comparison returns false. For example, the comparison, [b < 3.4] in the following statement returns false because the string "N/A" cannot be successfully cast to xs:double:

```
Xquery let $doc := <a><b>N/A</b></a> return $doc[b < 3.4];
```

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 first element node contains the string value "1", the second element "2", and the third element "2.0". 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)
```

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"]
  ```

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

Table 28. Results of logical expressions based on EBVs of operands (continued)

EBV of operand 1	Operator	EBV of operand 2	Result
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
  ```

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 in the previous
example:

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

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 { and } 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
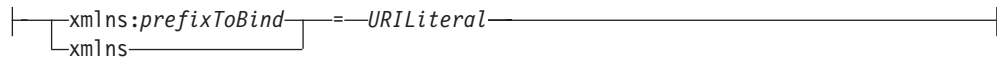error.

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
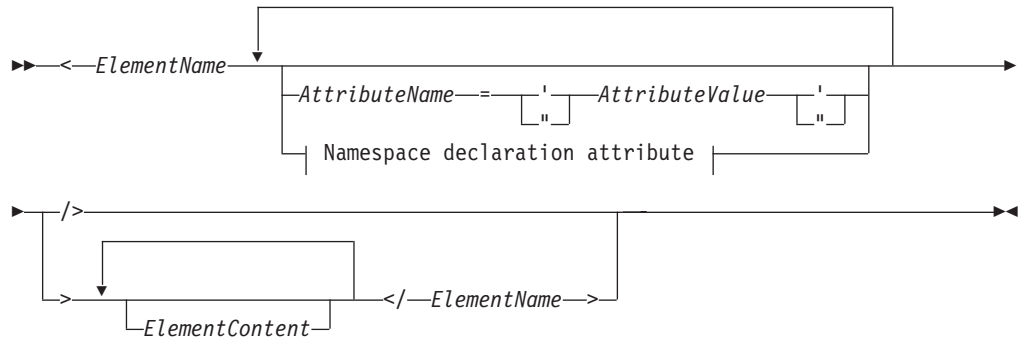
```
►►──<──ElementName──┬─────────────────────────────────────────────────────┬──►
                    │  ┌─────────────────────────────────────────────┐     │
                    └──┬─AttributeName──=──┬─'─┬──AttributeValue──┬─'─┬──┘  │
                       │                   └─"─┘                  └─"─┘      │
                       └─| Namespace declaration attribute |───────────────┘

►─┬─/>──────────────────────────────────────────────────────┬──►◄
  │                                                          │
  └─>──┬──────────────────┬──</──ElementName──>──────────────┘
       └──ElementContent──┘
```

Namespace declaration attribute:

```
├──┬─xmlns:prefixToBind─┬──=──URILiteral──────────────────────────────┤
   └─xmlns──────────────┘
```

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 a 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 (}})
<	<
&	&
"	" or two double quotation marks ("")
'	' 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 (}})
<	<
&	&

- **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>`

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>
```

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:
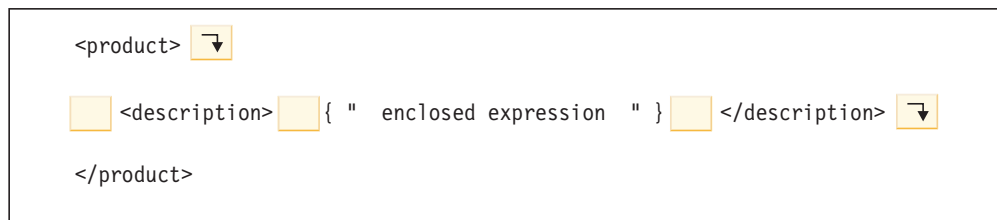
The boundary whitespace in this example includes the following characters: a newline character and four space characters that occur between the start tags of the `product` and `description` elements; four space characters that occur between the start tag of the `description` element and the enclosed 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, ` `) 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:

```
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 :

```
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> 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>      &#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>
```

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.

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.

Example

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.

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.

Example

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>`.

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

```
►►──attribute──AttributeName──{──────────────────────}────────────►◄
                                └─AttributeValueExpression─┘
```

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}
```

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.

Example

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>
}
```

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"}
```

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.

Direct processing instruction constructors
Direct processing instruction constructors use an XML-like notation to create processing instruction nodes.

Syntax

```
►►─<?─PITarget────────────────?>────────────────────◄◄
              └─DirPIContents─┘
```

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" ?>
```

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─PITarget─{─────────────────}──────────◄◄
                                     └─PIContentExpression─┘
```

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"}
```

Comment constructors

Comment constructors create comment nodes. XQuery provides both direct and computed constructors for creating comment nodes.

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.

Example

The following constructor creates a comment node:
```
<!-- This is an XML comment. -->
```

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.

Example

The following computed constructor creates the comment `<!--Houston, we have a problem.-->`:
```
let $homebase := "Houston"
return comment {fn:concat($homebase, ", we have a problem.")}
```

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 ──┘  └──where──Expression──┘

►────┬──────────────────────────────────────┬──return──Expression──►◄
     │              ┌──,──────────────┐      │
     │              │    ▼    ┌─ascending─┐  │
     └──order by────┴─Expression─┴─────────┴─┤
                                  └─descending─┘
```

for clause:

```
                  ┌──,──────────────────────────────────┐
                  │   ▼                                  │
├──for──┬─$VariableName──┬──────────────────────────┬──in──Expression──┴──┤
                         └─at──$PositionalVariableName─┘
```

let clause:

```
                  ┌──,────────────────────────────┐
                  │   ▼                            │
├──let──┬─$VariableName──:=──Expression──┴──────────────────────────────┤
```
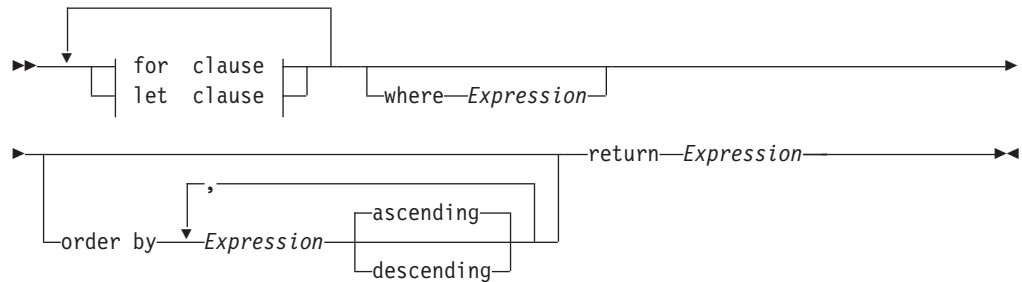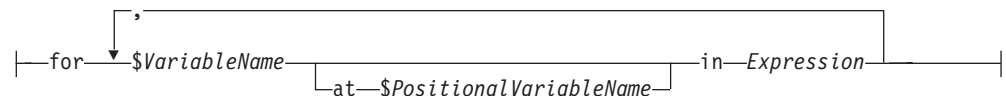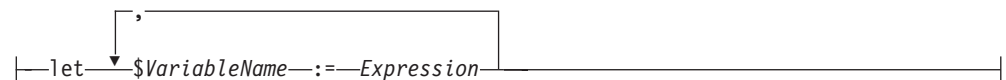
for
> The keyword that begins a **for** clause. A **for** clause iterates over the result of *Expression* and binds *VariableName* to each item that is returned by *Expression*.

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. If the **return** clause contains a non-updating expression, the FLWOR expression is a non-updating expression. 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.
>
> If the **return** clause contains an updating expression, the FLWOR expression is an updating expression. An updating FLWOR expression must be specified within the **modify** clause of a transform expression. The result of the updating FLWOR expression is a list of updates. The containing transform expression performs the updates after merging them with other updates returned by other updating expressions within the **modify** clause of the transform expression.

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.

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. In the following example, the binding sequence evaluates to an empty sequence and no iteration is performed. The node sequence in the **return** clause is not returned.

```
for $node in (<a test = "b" />, <a test = "c" />, <a test = "d" /> )[@test = "1"]
return
  <test>
    Sample return response
  </test>
```

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.

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.

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>
```

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 using for	`for $i in ("a", "b", "c")`
	`return <output>{$i}</output>`
		`<output>a</output>`
		`<output>b</output>`
		`<output>c</output>`

Bind a single variable using let	`let $i := ("a", "b", "c")` `return <output>{$i}</output>`	`<output>a b c</output>`

Bind multiple variables using for	`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>`

Bind multiple variables using let	`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.

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.

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.

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 as strings, 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 $prod in db2-fn:xmlcolumn('PRODUCT.DESCRIPTION')/product/description
    order by xs:decimal($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.DESCRIPTION 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 of the price element is xdt:untypedAtomic. Without this conversion, the result would use string ordering rather than numeric ordering.

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 $prod in db2-fn:xmlcolumn('PRODUCT.DESCRIPTION')/product
order by xs:decimal($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.

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 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.

If the expression in the **return** clause is a non-updating expression, the results of all the **return** clause evaluations are concatenated to form the result of the non-updating FLWOR expression.

If the expression in the **return** clause is an updating expression, the result of all the **return** clause evaluations is a list of updates. The transform expression that contains the FLWOR expression performs the updates after merging them with updates returned by other updating expressions within the **modify** clause of the transform expression.

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.

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.

```
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.DESCRIPTION')
    /product[@pid = $parts]/description/name }
</ProductList>
```

The query returns the following result:

```
<ProductList 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 PoNum="5003">
    <name>Snow Shovel, Basic 22 inch</name>
</ProductList>
<ProductList 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.

```
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 query returns the following result:

```
<city>Aurora
    <name>Robert Shoemaker</name>
</city>
<city>Markham
    <name>Kathy Smith</name>
    <name>Jim Noodle</name>
</city>
<city>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.

```
<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 query returns the following result:

```
<html>
<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 table:

Table 32. PO totals

PO Number	Status	Revenue
5004	Shipped	139.94
5001	Shipped	123.96
5003	Unshipped	9.99

FLWOR expression that updates XML data

The following example uses the CUSTOMER table from the DB2 SAMPLE database. In the CUSTOMER table, the XML column INFO contains customer address and phone information.

The transform expression creates a copy of an XML document containing customer information. In the **modify** clause, the FLWOR expression and the rename expression change all instances of the node name phone to the name phonenumber:

```
xquery
transform
  copy $mycust := db2-fn:sqlquery('select info from customer where cid = 1003')
  modify
    for $phone in $mycust/customerinfo/phone
    return
      do rename $phone as "phonenumber"
  return $mycust
```

When run against the SAMPLE database, the expression changes the node name phone to phonenumber and returns the following result:

```
<customerinfo Cid="1003">
  <name>Robert Shoemaker</name>
  <addr country="Canada">
    <street>1596 Baseline</street>
    <city>Aurora</city>
    <prov-state>Ontario</prov-state>
    <pcode-zip>N8X 7F8</pcode-zip>
  </addr>
  <phonenumber type="work">905-555-7258</phonenumber>
  <phonenumber type="home">416-555-2937</phonenumber>
  <phonenumber type="cell">905-555-8743</phonenumber>
  <phonenumber type="cottage">613-555-3278</phonenumber>
</customerinfo>
```

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.

> If either the **then** or **else** condition branch contains an updating expression, then the conditional expression is an updating expression. An updating expression must be within the **modify** clause of a transform expression.

For an updating conditional expression, each branch must contain either an updating expression or an empty sequence. Based on the value of the test expression, either the **then** or **else** clause is selected and evaluated. The result of the conditional updating expression is a list of updates returned by the selected branch. The containing transform expression performs the updates after merging them with updates returned by other updating expressions within the **modify** clause of the transform expression.

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:

```
for $prod in db2-fn:xmlcolumn('PRODUCT.DESCRIPTION')/product/description
return (
if (xs:decimal($prod/price) < 10)
  then <product basic = "true">{fn:data($prod/name)}</product>
  else <product basic = "false">{fn:data($prod/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.

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──┬─▼─$VariableName─in─Expression─┴─satisfies─TestExpression───────►◄
   └─every─┘
```
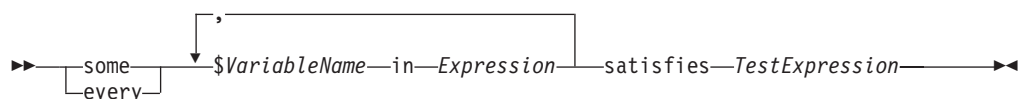
some
   When this keyword is specified, the quantified expression returns true if the effective boolean value of *TestExpression* is true for *at least one* item that is returned by *Expression*. Otherwise, the quantified expression returns false.

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
```

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 24. When casting a value to a data type, you can use the castable expression to test whether the value can be cast to the data type.

An empty sequence is a valid input value only when the target type is followed by a question mark (?).

If the target type of a cast expression is xs:QName or is a type derived from xs:QName or xs:NOTATION, and input expression is of type xs:string but it is not a literal string, an error is returned.

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.DESCRIPTION')/product/description/price
return $price cast as xs:decimal
```

When run against the PRODUCT.DESCRIPTION table of the SAMPLE database, the query in the example returns the following result:

```
9.99
19.99
49.99
3.99
```

Castable expressions

Castable expressions test whether a value can be cast to a specific data type. If the value can be cast to the data type, the castable expression returns true. Otherwise, the expression returns false.

Castable expressions can be used as predicates to avoid cast errors at evaluation time. They can also be used to select an appropriate type for processing a value. For information about which types can be cast to which other types, see "Type casting" on page 24.

Syntax

```
►►──Expression──castable as──TargetType───────────────────────────────►◄
                                        └──?──┘
```

Expression
> An XQuery expression that returns a single atomic value or an empty sequence.

TargetType
> The type used to test if the value of *Expression* can be cast. *TargetType* must be an atomic type that is one of the predefined XML schema types. The data types xs:NOTATION, xdt:anyAtomicType, and xs:anySimpleType are not valid types for *TargetType*.

?
> Indicates that an empty sequence is considered a valid instance of the target type. If *Expression* evaluates to an empty sequence and ? is not specified, the castable expression returns False.

Returned value

If *Expression* can be cast to *TargetType*, the castable expression returns true. Otherwise, the expression returns false.

If the result of *Expression* is an empty sequence and the question mark indicator follows *TargetType*, the castable expression returns true. In the following example, the question mark indicator follows the target type xs:integer.

```
$prod/revision castable as xs:integer?
```

If *TargetType* of a castable expression is xs:QName, or a type derived from xs:QName or xs:NOTATION, and *Expression* is of type xs:string but it is not a literal string, the returned value of the castable expression is false.

If the result of *Expression* is a sequence of more than one atomic value, an error is returned.

Examples

The following example uses the castable expression as a predicate to avoid errors at evaluation time. The following example avoids a dynamic error if @OrderDate is not a valid date.

```
$po/orderID[if ( $po/@OrderDate castable as xs:date)
   then xs:date($po/@OrderDate) gt xs:date("2000-01-01")
   else false()]
```

The predicate is true and returns the orderID only if the date attribute is a valid date greater than January 1, 2000. Otherwise, the predicate is false and returns an empty sequence.

The following example uses the castable expression to select an appropriate type for processing of a given value. The example uses castable to cast a postal code as either an integer or a string:

```
if ($postalcode castable as xs:integer)
   then $postalcode cast as xs:integer
   else $postalcode cast as xs:string
```

The following example uses the castable expression in the FLWOR **let** clause to test the value of $prod/mfgdate and bind a value to $currdate. The castable expression and the cast expression support processing an empty sequence using the question mark indicator.

```
let $currdate := if ($prod/mfgdate castable as xs:date?)
    then $prod/mfgdate cast as xs:date?
    else "1000-01-01" cast as xs:date
```

If the value of $prod/mfgdate can be cast as xs:date, it is cast to the data type and is bound to $currdate. If $prod/mfgdate is an empty sequence, an empty sequence is bound to $currdate. If $prod/mfgdate cannot be cast as xs:date, a value of 1000-01-01 of type xs:date is bound to $currdate.

The following example uses the castable expression to test the value of the product category before performing a comparison. In the XML column FEATURES.INFO, the documents contain the element /prod/category. The value is either a numeric code or string code. The castable expressions in the XMLEXISTS predicate tests the value of /prod/category before performing a comparison to avoid errors at evaluation time.

```
SELECT F.PRODID FROM F FEATURES
WHERE xmlexists('$test/prod/category[ (( . castable as xs:double) and . > 100 ) or
    (( . castable as xs:string) and . > "A100" )]'
    passing F.INFO as "test")
```

The returned values are product IDs where the category codes are either greater than the number 100 or greater than the string "A100."

Transform expression and updating expressions

To update existing XML data with DB2 XQuery, use updating expressions within the **modify** clause of a transform expression.

Use of updating expressions in a transform expression

DB2 XQuery updating expressions must be used in the **modify** clause of a transform expression. The updating expressions operate on the copied nodes created by the **copy** clause of the transform expression.

The following expressions are updating expressions:
- A delete expression
- An insert expression
- A rename expression
- A replace expression
- A FLWOR expression that contains an updating expression in its **return** clause
- A conditional expression that contains an updating expression in its **then** or **else** clause
- Two or more updating expressions, separated by commas where all operands are either updating expressions or an empty sequence

DB2 XQuery returns an error for updating expressions that are not valid. For example, DB2 XQuery returns an error if one branch of a conditional expression contains an updating expression and the other branch contains a non-updating expression that is not the empty sequence.

A transform expression is not an updating expression, because it does not modify any existing nodes. A transform expression creates modified copies of existing nodes. The result of a transform expression can include nodes created by updating expressions in the **modify** clause of the transform expression and copies of previously existing nodes.

Processing XQuery updating operations

In a transform expression, the **modify** clause can specify multiple updates. For example, the **modify** clause can contain two updating expressions, one expression that replaces an existing value, and the other expression that inserts a new element. When the **modify** clause contains multiple updating expressions, each updating expression is evaluated independently and results in a list of change operations associated with specific nodes that were created by the **copy** clause of the transform expression.

Within a **modify** clause, updating expressions cannot modify new nodes that are added by other updating expressions. For example, if an updating expression adds a new element node, another updating expression cannot change the node name of the newly created node.

All the change operations specified in the **modify** clause of the transform expression are collected and effectively applied in the following order:

1. The following updating operations are performed in a nondeterministic order:
   - Insert operations that do not use ordering keywords such as **before**, **after**, **as first**, or **as last**.
   - All rename operations.
   - Replace operations where the keywords **value of** are specified and the target node is an attribute, text, comment, or processing instruction node.
2. Insert operations that use ordering keywords such as **before**, **after**, **as first**, or **as last**.
3. Replace operations where the keywords **value of** are not specified.
4. Replace operations where the keywords **value of** are specified and the target node is an element node.
5. All delete operations.

The order in which change operations are applied ensures that a series of multiple changes will have a deterministic result. For an example of how the order of update operations guarantees that a series of multiple changes will have a deterministic result, see the last XQuery expression in "Examples" on page 113.

Invalid XQuery updating operations

During processing of a transform expression, DB2 XQuery returns an error if any of the following conditions occur:
- Two or more rename operations are applied to the same node.
- Two or more replace operations that use the **value of** keywords are applied to the same node.
- Two or more replace operations that don't use the **value of** keywords are applied to the same node.
- The result of the transform expression is not a valid XDM instance.

  An example of an invalid XDM instance is one that contains an element with two attributes where both attributes have the same name.

- The XDM instance contains inconsistent namespace bindings.

  The following are examples of inconsistent namespace bindings:

  – A namespace binding in the QName of an attribute node does not agree with the namespace bindings in its parent element node.

  – The namespace bindings in two attribute nodes with the same parent do not agree with each other.

Examples

In the following example, the **copy** clause of a transform expression binds the variable $product to a copy of an element node, and the **modify** clause of the transform expression uses two updating expressions to change the copied node:

```
xquery
transform
copy $product := db2-fn:sqlquery(
  "select description from product where pid='100-100-01'")/product
modify(
  do replace value of $product/description/price with 349.95,
  do insert <status>Available</status> as last into $product )
return $product
```

The following example uses an XQuery transform expression within an SQL UPDATE statement to modify XML data in the CUSTOMER table. The SQL UPDATE statement operates on a row of the CUSTOMER table. The transform expression creates a copy of the XML document from the INFO column of the row, and adds a status element to the copy of the document. The UPDATE statement replaces the document in the INFO column of the row with the copy of the document modified by the transform expression:

```
UPDATE customer
SET info = xmlquery( 'transform
   copy $newinfo := $info
   modify do insert <status>Current</status> as last into $newinfo/customerinfo
   return $newinfo' passing info as "info")
WHERE cid = 1003
```

The following examples use the CUSTOMER table from the DB2 SAMPLE database. In the CUSTOMER table, the XML column INFO contains customer address and phone information.

In the following example, the SQL SELECT statement operates on a row of the CUSTOMER table. The **copy** clause of the transform expression creates a copy of the XML document from the column INFO. The delete expression deletes address information, and non-work phone numbers, from the copy of the document. The **return** uses the customer ID attribute and country attribute from the original document from the CUSTOMER table:

```
SELECT XMLQUERY( 'transform
  copy $mycust := $d
  modify
    do delete ( $mycust/customerinfo/addr,
      $mycust/customerinfo/phone[@type != "work"] )
  return
  <custinfo>
   <Cid>{data($d/customerinfo/@Cid)}</Cid>
   {$mycust/customerinfo/*}
   <country>{data($d/customerinfo/addr/@country)}</country>
  </custinfo>'
  passing INFO as "d")
FROM CUSTOMER
WHERE CID = 1003
```

When run against the SAMPLE database, the statement returns the following result:

```
<custinfo>
  <Cid>1003</Cid>
  <name>Robert Shoemaker</name>
  <phone type="work">905-555-7258</phone>
  <country>Canada</country>
</custinfo>
```

In the following example, the XQuery expression demonstrates how the order of update operations guarantees that a series of multiple changes will have a deterministic result. The insert expression adds a `status` element after a `phone` element, and the replace expression replaces the `phone` element with an `email` element:

```
xquery
let $email  := <email>jnoodle@my-email.com</email>
let $status := <status>current</status>
return
  transform
  copy $mycust := db2-fn:sqlquery('select info from customer where cid = 1002')
  modify (
    do replace $mycust/customerinfo/phone with $email,
    do insert $status after $mycust/customerinfo/phone[@type = "work"] )
  return $mycust
```

In the **modify** clause, the replace expression is before the insert expression. However, when updating the copied node sequence `$mycust`, the insert update operation is performed before the replace update operation to ensure a deterministic result. When run against the SAMPLE database, the expression returns the following result:

```
<customerinfo Cid="1002">
  <name>Jim Noodle</name>
  <addr country="Canada">
    <street>25 EastCreek</street>
    <city>Markham</city>
    <prov-state>Ontario</prov-state>
    <pcode-zip>N9C 3T6</pcode-zip>
  </addr>
  <email>jnoodle@my-email.com</email>
  <status>current</status>
</customerinfo>
```

If the replace operation were performed first, the `phone` element would not be in the node sequence, and the operation to insert the `status` element after the `phone` element would have no meaning.

For information about the order of update operations, see "Processing XQuery updating operations" on page 112.

Transform expression

A transform expression creates copies of one or more nodes. Updating expressions in the **modify** clause of the transform expression change the copied nodes. The expression in the **return** clause specifies the result of the transform expression.

Syntax
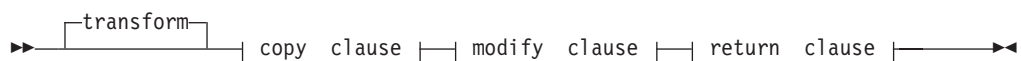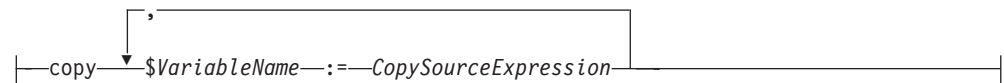
```
                ┌─transform─┐
►►──────────────┴───────────┴──┤ copy  clause ├──┤ modify  clause ├──┤ return  clause ├────────►◄
```

copy clause:

```
         ┌──────,───────────────────────────┐
├──copy──▼──$VariableName──:=──CopySourceExpression──┴──────────────────────┤
```

modify clause:

```
├──modify──ModifyExpression──────────────────────────────────────────────┤
```

return clause:

```
├──return──ReturnExpression──────────────────────────────────────────────┤
```

Parameters

transform
   Optional keyword that can be used to begin a transform expression.

copy
   Keyword that begins the **copy** clause of a transform expression. Each
   VariableName in the **copy** clause is bound to a logical copy of the node tree that
   is returned by the corresponding *CopySourceExpression*.

VariableName
   The name of the variable to bind to a copy of the node tree returned by
   CopySourceExpression.

CopySourceExpression
   An XQuery expression that is not an updating expression. The expression must
   return a single node, together with its descendants (if any), called a *node tree*.

   If the expression includes a top-level comma operator, the expression must be
   enclosed in parentheses. The *CopySourceExpression* is evaluated as though it
   were an enclosed expression in an element constructor.
   The nodes created by the **copy** clause have new node identities and are
   untyped.

modify
   Keyword that begins the **modify** clause of a transform expression.

ModifyExpression
   An updating expression or an empty sequence. If the expression includes a
   top-level comma operator, the expression must be enclosed in parentheses. The
   updating expression is evaluated and the resulting updates are applied to
   nodes created by the **copy** clause.

   DB2 XQuery returns an error if the target node of an updating expression is
   not a node created by the **copy** clause of the containing transform expression.
   For example, DB2 XQuery returns an error if a rename expression tries to
   rename a node that is not created by the **copy** clause.

   The updates specified in a **modify** clause can result in a node that has multiple,
   adjacent text nodes among its children. If a node has multiple, adjacent text
   nodes among its children, the adjacent text nodes are merged into a single text
   node. The string value of the resulting text node is the concatenated string

values of the adjacent text nodes with no intervening spaces. If a child node is created that is a text node with a string value that is a zero-length string, the text node is deleted.

return
> The keyword that begins the **return** clause of a transform expression.

ReturnExpression
> An XQuery expression that is not an updating expression. If the expression includes a top-level comma operator, the expression must be enclosed in parentheses.
>
> The expression in the **return** clause is evaluated and is returned as the result of the transform expression. Expressions in the **return** clause can access the nodes changed or created by updating expressions in the **modify** clause.
>
> The **return** clause of a transform expression is not restricted to return only nodes that were created by the **copy** clause. The *ReturnExpression* can return any combination of copied nodes, original nodes, and constructed nodes.

Examples

The following examples use the CUSTOMER table from the DB2 SAMPLE database. In the CUSTOMER table, the XML column INFO contains customer address and phone information.

In the following example, the **copy** clause of the transform expression creates a copy of the XML document from the column INFO. In the **modify** clause, the delete expression deletes all phone numbers from the XML document where the phone's type attribute is not home:

```
xquery
transform
   copy $mycust := db2-fn:sqlquery('select INFO from CUSTOMER where Cid = 1003')
   modify
     do delete $mycust/customerinfo/phone[@type!="home"]
   return $mycust;
```

When run against the SAMPLE database, the expression returns the following result:

```
<customerinfo Cid="1003">
  <name>Robert Shoemaker</name>
  <addr country="Canada">
    <street>1596 Baseline</street>
    <city>Aurora</city>
    <prov-state>Ontario</prov-state>
    <pcode-zip>N8X 7F8</pcode-zip>
  </addr>
  <phone type="home">416-555-2937</phone>
</customerinfo>
```

The following expression does not use the optional keyword **transform**. The transform expression starts with the **copy** clause and is equivalent to the previous expression.

```
xquery
copy $mycust := db2-fn:sqlquery('select INFO from CUSTOMER where Cid = 1003')
modify
   do delete $mycust/customerinfo/phone[@type!="home"]
return $mycust;
```

In the following example, the SQL UPDATE statement modifies and validates the XML document from a row of the CUSTOMER table. The **copy** clause of the transform expression creates a copy of the XML document from the column INFO. The replace expression changes the value of the name element in the copy of the XML document. The copy of the document is not validated. The XMLVALIDATE function validates the document copy:

```
UPDATE customer set info = XMLVALIDATE(
  XMLQUERY('transform
  copy $mycust := $cust
  modify
    do replace value of $mycust/customerinfo/name with "Larry Menard, Jr."
  return $mycust'
  passing info as "cust" )
ACCORDING TO XMLSCHEMA ID customer)
where cid = 1005
```

Basic updating expressions

Using the four basic XQuery updating expressions, you can create complex updating expressions to update existing XML data. When using DB2 XQuery, updating expressions are used within the **modify** clause of a transform expression.

Delete expression

A delete expression deletes zero or more nodes from a node sequence.

Syntax

▶▶──do delete──*TargetExpression*──▶◀

do delete
>   The keywords that begin a delete expression.

TargetExpression
>   An XQuery expression that is not an updating expression. If the expression includes a top-level comma operator, the expression must be enclosed in parentheses. The result of *TargetExpression* must be a sequence of zero or more nodes. Each node's parent property cannot be empty.

The transform expression evaluates the delete expression and generates a list of updates that consist of nodes to be deleted. Any node that matches the *TargetExpression* is marked for deletion. When deleting the *TargetExpression* nodes, the nodes are detached from their parent nodes. The nodes and the nodes' children are no longer part of the node sequence.

Examples

The following examples use the CUSTOMER table from the DB2 SAMPLE database. In the CUSTOMER table, the XML column INFO contains customer address and phone information.

The following expression deletes the address element and all of its children nodes, and all phone numbers from the XML document where the phone's attribute type is not home:

```
xquery
transform
copy $mycust := db2-fn:sqlquery('select INFO from CUSTOMER where Cid =1003')
modify
  do delete ($mycust/customerinfo/addr, $mycust/customerinfo/phone[@type!="home"])
return $mycust
```

When run against the SAMPLE database, the expression returns the following
result:

```
<customerinfo Cid="1003">
  <name>Robert Shoemaker</name>
  <phone type="home">416-555-2937</phone>
</customerinfo>
```

The following example deletes the type attribute from any phone element node
when the attribute value is home.

```
xquery
transform
copy $mycust := db2-fn:sqlquery('select info from customer where cid = 1004')
modify (
  for $phone in $mycust/customerinfo//phone[@type="home"]
  return
    do delete $phone/@type )
return $mycust
```

When run against the SAMPLE database, the expression returns the following
result:

```
<customerinfo Cid="1004">
  <name>Matt Foreman</name>
  <addr country="Canada">
    <street>1596 Baseline</street>
    <city>Toronto</city>
    <prov-state>Ontario</prov-state>
    <pcode-zip>M3Z 5H9</pcode-zip>
  </addr>
  <phone type="work">905-555-4789</phone>
  <phone>416-555-3376</phone>
  <assistant><name>Gopher Runner</name>
    <phone>416-555-3426</phone>
  </assistant>
</customerinfo>
```

The expression deletes the type attribute from both the customer's phone number
and the assistant's phone number.

Insert expression

An insert expression inserts copies of one or more nodes into a designated position
in a node sequence.

Syntax

```
►►──do insert──SourceExpression──┬─before────────┬──TargetExpression──────────────►◄
                                 ├─after─────────┤
                                 ├─as first into─┤
                                 ├─as last into──┤
                                 └─into──────────┘
```
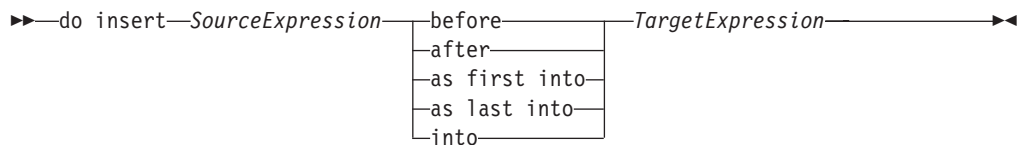
do insert
>   The keywords that begin an insert expression.

SourceExpression

An XQuery expression that is not an updating expression. If the expression includes a top-level comma operator, then the expression must be enclosed in parentheses. The expression is evaluated as though it were an enclosed expression in an element constructor. The result of the *SourceExpression* is a sequence of zero or more nodes to be inserted, called the *insertion sequence*. If the insertion sequence contains a document node, the document node is replaced in the insertion sequence by its children.

If the insertion sequence contains attribute nodes that appear first in the sequence, the attributes are added to the *TargetExpression* node or to its parent, depending on the keyword specified. If the insertion sequence contains an attribute node following a node that is not an attribute node, DB2 XQuery returns an error.

before

Keyword that specifies the *SourceExpression* nodes become the preceding siblings of the *TargetExpression* node.

The *SourceExpression* nodes are inserted immediately before the *TargetExpression* node. If multiple nodes are inserted before the *TargetExpression*, they are inserted in nondeterministic order, but the set of inserted nodes appear immediately before the *TargetExpression*. If the insertion sequence contains attribute nodes that appear first in the sequence, the attribute nodes become attributes of the parent of the target node.

after

Keyword that specifies the *SourceExpression* nodes become the following siblings of the *TargetExpression* node.

The *SourceExpression* nodes are inserted immediately after the *TargetExpression* node. If multiple nodes are inserted after the *TargetExpression*, they are inserted in nondeterministic order, but the set of inserted nodes appear immediately after the *TargetExpression*. If the insertion sequence contains attribute nodes that appear first in the sequence, the attribute nodes become attributes of the parent of the target node.

as first into

Keywords that specify the *SourceExpression* nodes become the first children of the *TargetExpression* node.

If multiple nodes are inserted as the first children of the *TargetExpression* node, they are inserted in nondeterministic order, but the set of inserted nodes appear as the first children of the *TargetExpression*. If the insertion sequence contains attribute nodes that appear first in the sequence, the attribute nodes become attributes of the target node.

as last into

Keywords that specify the *SourceExpression* nodes become the last children of the *TargetExpression* node.

If multiple nodes are inserted as the last children of the *TargetExpression* node, they are inserted in nondeterministic order, but the set of inserted nodes appear as the last children of the *TargetExpression* node. If the insertion sequence contains attribute nodes that appear first in the sequence, the attribute nodes become attributes of the target node.

into

Keyword that specifies the *SourceExpression* nodes become the children of the *TargetExpression* node in a nondeterministic order.

The *SourceExpression* nodes are inserted as children of the *TargetExpression* node in nondeterministic positions. If the insertion sequence contains attribute nodes that appear first in the sequence, the attribute nodes become attributes of the target node.

TargetExpression

An XQuery expression that is not an updating expression. If the expression includes a top-level comma operator, then the expression must be enclosed in parentheses. Based on the keywords specified before the *TargetExpression*, the following rules apply:

- If **before** or **after** is specified, the result of *TargetExpression* must be an element, text, processing instruction, or comment node whose parent property is not empty. If the parent of the *TargetExpression* node is a document node and **before** or **after** is specified, the insertion sequence cannot contain an attribute node.

- If **into**, **as first into**, or **as last into** is specified, the result of *TargetExpression* must be a single element node or a single document node.

- If **into** is specified and *TargetExpression* is a document node, the insertion sequence cannot contain an attribute node.

Examples

The following examples use the CUSTOMER table from the DB2 SAMPLE database. In the CUSTOMER table, the XML column INFO contains customer address and phone information.

In the following example, the **copy** clause of the transform expression creates a copy of the XML document from column INFO. The insert expression inserts the `billto` element and all its children after the last `phone` element :

```
xquery
  transform
  copy $mycust := db2-fn:sqlquery('select info from customer where cid = 1004')
  modify
    do insert
      <billto country="Canada">
          <street>4441 Wagner</street>
          <city>Aurora</city>
          <prov-state>Ontario</prov-state>
          <pcode-zip>N8X 7F8</pcode-zip>
      </billto>
    after $mycust/customerinfo/phone[last()]
  return $mycust
```

When run against the SAMPLE database, the expression returns the following result:

```
<customerinfo Cid="1004">
  <name>Matt Foreman</name>
  <addr country="Canada">
    <street>1596 Baseline</street>
    <city>Toronto</city>
    <prov-state>Ontario</prov-state>
    <pcode-zip>M3Z 5H9</pcode-zip>
  </addr>
  <phone type="work">905-555-4789</phone>
  <phone type="home">416-555-3376</phone>
  <billto country="Canada">
    <street>4441 Wagner</street>
    <city>Aurora</city>
    <prov-state>Ontario</prov-state>
    <pcode-zip>N8X 7F8</pcode-zip>
```

```
    </billto>
  <assistant>
    <name>Gopher Runner</name>
    <phone type="home">416-555-3426</phone>
  </assistant>
</customerinfo>
```

The following example inserts the attribute `extension` with the value `x2334` into the phone element where the phone's type attribute is `work`:

```
xquery
let $phoneext :=  attribute extension { "x2334" }
return
  transform
  copy $mycust := db2-fn:sqlquery('select info from customer where cid = 1004')
  modify
    do insert  $phoneext into $mycust/*:customerinfo/*:phone[@type="work"]
  return $mycust
```

When run against the SAMPLE database, the expression returns the following result:

```
<customerinfo Cid="1004">
  <name>Matt Foreman</name>
    <addr country="Canada">
      <street>1596 Baseline</street>
      <city>Toronto</city>
      <prov-state>Ontario</prov-state>
      <pcode-zip>M3Z 5H9</pcode-zip>
    </addr>
  <phone extension="x2334" type="work">905-555-4789</phone>
  <phone type="home">416-555-3376</phone>
  <assistant>
    <name>Gopher Runner</name>
    <phone type="home">416-555-3426</phone>
  </assistant>
</customerinfo>
```

Rename expression

A rename expression replaces the name property of a data model node with a new QName.

Syntax

►►──do rename──*TargetExpression*──────as──────*NewNameExpression*───────────────────►◄

do rename
> The keywords that begin a rename expression.

TargetExpression
> An XQuery expression that is not an updating expression. The result of *TargetExpression* must be a single element, attribute, or processing instruction node. If the expression includes a top-level comma operator, then the expression must be enclosed in parentheses.
>
> The rename expression affects only the *TargetExpression* node. If the *TargetExpression* node is an element node, the expression has no effect on any attributes or children of the target node. If the *TargetExpression* node is an attribute node, the expression has no effect on other attributes or descendants of the parent node of the *TargetExpression* node.

as The keyword that begins the new name expression.

NewNameExpression

An XQuery expression that is not an updating expression. The result of *NewNameExpression* must be a value of type xs:string, xs:QName, xs:untypedAtomic, or a node from which such a value can be extracted by the atomization process. If the expression includes a top-level comma operator, then the expression must be enclosed in parentheses.

The resulting value is converted to a QName, resolving its namespace prefix, if any, according to the statically known namespaces. The result is either an error or an expanded QName. The expanded QName replaces the name of the *TargetExpression* node.

If the new QName contains the same prefix but a different URI than an in-scope namespace of the *TargetExpression* node, DB2 XQuery returns an error.

Examples

The following examples use the CUSTOMER table from the DB2 SAMPLE database. In the CUSTOMER table, the XML column INFO contains customer address and phone information.

In the following example, the **copy** clause of the transform expression creates a copy of the XML document from column INFO. The rename expression changes the name property of the addr element to shipto:

```
xquery
transform
copy $mycust := db2-fn:sqlquery('select info from customer where cid = 1000')
modify
  do rename $mycust/customerinfo/addr as "shipto"
return $mycust
```

When run against the SAMPLE database, the expression returns the following result:

```
<customerinfo Cid="1000">
  <name>Kathy Smith</name>
  <shipto country="Canada">
    <street>5 Rosewood</street>
    <city>Toronto</city>
    <prov-state>Ontario</prov-state>
    <pcode-zip>M6W 1E6</pcode-zip>
  </shipto>
  <phone type="work">416-555-1358</phone>
</customerinfo>
```

In the following example, the **modify** clause of the transform expression contains a FLWOR expression and a rename expression that changes the name property of all instances of the element phone to phonenumber:

```
xquery
transform
copy $mycust :=  db2-fn:sqlquery('select info from customer where cid = 1003')
modify
  for $phone in $mycust/customerinfo/phone
  return
    do rename $phone as "phonenumber"
return $mycust
```

When run against the SAMPLE database, the expression returns the following result:

```
<customerinfo Cid="1003">
  <name>Robert Shoemaker</name>
  <addr country="Canada">
    <street>1596 Baseline</street>
    <city>Aurora</city>
    <prov-state>Ontario</prov-state>
    <pcode-zip>N8X 7F8</pcode-zip>
  </addr>
  <phonenumber type="work">905-555-7258</phonenumber>
  <phonenumber type="home">416-555-2937</phonenumber>
  <phonenumber type="cell">905-555-8743</phonenumber>
  <phonenumber type="cottage">613-555-3278</phonenumber>
</customerinfo>
```

In the following example, the rename expression changes the name of the `addr` element's attribute from `country` to `geography`:

```
xquery
transform
copy $mycust := db2-fn:sqlquery('select info from customer where cid = 1000')
modify
  do rename $mycust/customerinfo/addr/@country as "geography"
return $mycust
```

When run against the SAMPLE database, the expression returns the following result:

```
<customerinfo Cid="1000">
  <name>Kathy Smith</name>
  <addr geography="Canada">
    <street>5 Rosewood</street>
    <city>Toronto</city>
    <prov-state>Ontario</prov-state>
    <pcode-zip>M6W 1E6</pcode-zip>
  </addr>
  <phone type="work">416-555-1358</phone>
</customerinfo>
```

The following example uses the rename expression and the fn:QName function to add the namespace prefix `other` to the names of the customer's non-work phone number elements and attributes. The prefix `other` is bound to the URI `http://otherphone.com`:

```
xquery
transform
copy $mycust := db2-fn:sqlquery('select info from customer where cid = 1004')
modify
  for $elem in $mycust/customerinfo/phone[@type != "work"]
  let $elemLocalName := fn:local-name($elem)
  let $newElemQName := fn:QName("http://otherphone.com", fn:concat("other:",
    $elemLocalName))
  return
    ( do rename $elem as $newElemQName,
    for $a in $elem/@* let $attrlocalname := fn:local-name($a)
    let $newAttrName := fn:QName("http://otherphone.com", fn:concat("other:",
      $attrlocalname))
    return
      do rename $a as $newAttrName )
return $mycust
```

When run against the SAMPLE database, the expression returns the following result:

```
<customerinfo Cid="1004">
  <name>Matt Foreman</name>
  <addr country="Canada">
    <street>1596 Baseline</street>
```

```
      <city>Toronto</city>
      <prov-state>Ontario</prov-state>
      <pcode-zip>M3Z 5H9</pcode-zip>
    </addr>
    <phone type="work">905-555-4789</phone>
    <other:phone xmlns:other="http://otherphone.com" other:type="home">
      416-555-3376</other:phone>
    <assistant>
      <name>Gopher Runner</name>
      <phone type="home">416-555-3426</phone>
    </assistant>
</customerinfo>
```

If you use the following expression in the transform expression's **return** clause, the phone element nodes that use the default element namespace appear as child nodes of the primary node, and the other:phone element node appears as the child node of the secondary node.

```
<phonenumbers xmlns:other="http://otherphone.com">
  <primary>
    { $mycust//phone }
  </primary>
  <secondary>
    { $mycust//other:phone }
  </secondary>
</phonenumbers>
```

When run against the SAMPLE database, the transform expression returns the following result:

```
<phonenumbers xmlns:other="http://otherphone.com"
  <primary>
    <phone type="work">905-555-4789</phone>
    <phone type="home">416-555-3426</phone>
  </primary>
  <secondary>
    <other:phone other:type="home">416-555-3376</other:phone>
  </secondary>
</phonenumbers>
```

Replace expression

A replace expression replaces an existing node with a new sequence of zero or more nodes, or replaces a node's value while preserving the node's identity.

Syntax

```
►►──do replace─┬──────────┬──TargetExpression────with────SourceExpression──────────►◄
               └─value of─┘
```

do replace
    The keywords that begin a replace expression.

TargetExpression
    An XQuery expression that is not an updating expression. If the expression includes a top-level comma operator, the expression must be enclosed in parentheses. The result of *TargetExpression* must be a single node that is not a document node. If the result of *TargetExpression* is a document node, DB2 XQuery returns an error.

    If the **value of** keywords are not specified, the result of *TargetExpression* must be a single node whose parent property is not empty.

value of
> The keywords that specify replacing the value of the *TargetExpression* node while preserving the node's identity.

with
> The keyword that begins the source expression.

SourceExpression
> An XQuery expression that is not an updating expression. If the expression includes a top-level comma operator, the expression must be enclosed in parentheses.
>
> If the **value of** keywords are specified, the *SourceExpression* is evaluated as though it were the content expression of a text node constructor. The result of the *SourceExpression* is a single text node or an empty sequence.
>
> If the **value of** keywords are not specified, the result of the *SourceExpression* must be a sequence of nodes. The *SourceExpression* is evaluated as though it were an expression enclosed in an element constructor. If the *SourceExpression* sequence contains a document node, the document node is replaced by its children. The *SourceExpression* sequence must consist of the following node types:
> - If the *TargetExpression* node is an attribute node, the replacement sequence must consist of zero or more attribute nodes.
> - If the *TargetExpression* node is an element, text, comment, or processing instruction node, the replacement sequence must consist of some combination of zero or more element, text, comment, or processing instruction nodes.

The following updates are generated when the **value of** keywords are specified:
- If the *TargetExpression* node is an element node, the existing children of the *TargetExpression* node are replaced by the text node returned by the *SourceExpression*. If the *SourceExpression* returns an empty sequence, the children property of the *TargetExpression* node becomes empty. If the *TargetExpression* node contains attribute nodes, they are not affected.
- If the *TargetExpression* node is not an element node, the string value of the *TargetExpression* node is replaced by the string value of the text node returned by the *SourceExpression*. If the *SourceExpression* does not return a text node, the string value of the *TargetExpression* node is replaced by a zero-length string.

The following updates are generated when the **value of** keywords are not specified:
- *SourceExpression* nodes replace the *TargetExpression* node. The parent node of the *TargetExpression* node becomes the parent of each of the *SourceExpression* nodes. The *SourceExpression* nodes occupy the position in the node hierarchy occupied by the *TargetExpression* node.
- The *TargetExpression* node, all its attributes and descendants are detached from the node sequence.

Examples

The following examples use the CUSTOMER table from the DB2 SAMPLE database. In the CUSTOMER table, the XML column INFO contains customer address and phone information.

In the following example, the **copy** clause of the transform expression creates a copy of the XML document from column INFO. The replace expression replaces the addr element and its children:

```
xquery
transform
copy $mycust := db2-fn:sqlquery('select info from customer where cid = 1000')
modify
  do replace $mycust/customerinfo/addr
    with
      <addr country="Canada">
        <street>1596 14th Avenue NW</street>
        <city>Calgary</city>
        <prov-state>Alberta</prov-state>
        <pcode-zip>T2N 1M7</pcode-zip>
      </addr>
return $mycust
```

When run against the SAMPLE database, the expression returns the following result with the replaced address information:

```
<customerinfo Cid="1000">
  <name>Kathy Smith</name>
  <addr country="Canada">
    <street>1596 14th Avenue NW</street>
    <city>Calgary</city>
    <prov-state>Alberta</prov-state>
    <pcode-zip>T2N 1M7</pcode-zip>
  </addr>
  <phone type="work">416-555-1358</phone>
</customerinfo>
```

The following expression replaces the value of the customer phone element's type attribute from home to personal:

```
xquery
transform
copy $mycust := db2-fn:sqlquery('select info from customer where cid = 1004')
modify
  do replace value of $mycust/customerinfo/phone[@type="home"]/@type with "personal"
return $mycust
```

When run against the SAMPLE database, the expression returns the following result with the replaced attribute value:

```
<customerinfo Cid="1004">
  <name>Matt Foreman</name>
  <addr country="Canada">
    <street>1596 Baseline</street>
    <city>Toronto</city>
    <prov-state>Ontario</prov-state>
    <pcode-zip>M3Z 5H9</pcode-zip>
  </addr>
  <phone type="work">905-555-4789</phone>
  <phone type="personal">416-555-3376</phone>
  <assistant>
    <name>Gopher Runner</name>
    <phone type="home">416-555-3426</phone>
  </assistant>
</customerinfo>
```

The value of the assistant's phone attribute has not been changed.

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 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.

DB2 XQuery functions by category

The following categories of DB2 XQuery functions are available: string, boolean, number, date and time, sequence, QName, node, and other.

String functions

Function	Description
"codepoints-to-string function" on page 142	The fn:codepoints-to-string function returns the string equivalent of a sequence of Unicode code points.
"compare function" on page 143	The fn:compare function compares two strings.
"concat function" on page 143	The fn:concat function returns a string that is the concatenation of two or more atomic values.
"contains function" on page 144	The fn:contains function determines whether a string contains a specific substring. The search string is matched using the default collation.
"ends-with function" on page 154	The fn:ends-with function determines whether a string ends with a specific substring. The search string is matched using the default collation.
"lower-case function" on page 164	The fn:lower-case function converts a string to lowercase.
"matches function" on page 165	The fn:matches function determines whether a string matches a specific pattern.

Function	Description
"normalize-space function" on page 176	The fn:normalize-space function strips leading and trailing white space characters from a string and replaces each internal sequence of white space characters with a single blank character.
"normalize-unicode function" on page 176	The fn:normalize-unicode function performs Unicode normalization on a string.
"replace function" on page 181	The fn:replace function compares each set of characters within a string to a specific pattern, and then it replaces the characters that match the pattern with another set of characters.
"starts-with function" on page 192	The fn:starts-with function determines whether a string begins with a specific substring. The search string is matched using the default collation.
"string function" on page 192	The fn:string function returns the string representation of a value.
"string-join function" on page 193	The fn:string-join function returns a string that is generated by concatenating items separated by a separator character.
"string-length function" on page 194	The fn:string-length function returns the length of a string.
"string-to-codepoints function" on page 194	The fn:string-to-codepoints function returns a sequence of Unicode code points that corresponds to a string value.
"substring function" on page 196	The fn:substring function returns a substring of a string.
"substring-after function" on page 196	The fn:substring-after function returns a substring that occurs in a string after the end of the first occurrence of a specific search string. The search string is matched using the default collation.
"substring-before function" on page 197	The fn:substring-before function returns a substring that occurs in a string before the first occurrence of a specific search string. The search string is matched using the default collation.
"tokenize function" on page 201	The fn:tokenize function breaks a string into a sequence of substrings.
"translate function" on page 202	The fn:translate function replaces selected characters in a string with replacement characters.
"upper-case function" on page 204	The fn:upper-case function converts a string to uppercase.

Boolean functions

Function	Description
"boolean function" on page 140	The fn:boolean function returns the effective Boolean value of a sequence.

Function	Description
"false function" on page 156	The fn:false function returns the xs:boolean value false.
"not function" on page 177	The fn:not function returns false if the effective boolean value of a sequence is true and returns true if the effective boolean value of a sequence is false.
"true function" on page 203	The fn:true function returns the xs:boolean value true.

Number functions

Function	Description
"abs function" on page 139	The fn:abs function returns the absolute value of a numeric value.
"avg function" on page 139	The fn:avg function returns the average of the values in a sequence.
"ceiling function" on page 141	The fn:ceiling function returns the smallest integer that is greater than or equal to a specific numeric value.
"floor function" on page 157	The fn:floor function returns the largest integer that is less than or equal to a specific numeric value.
"max function" on page 166	The fn:max function returns the maximum of the values in a sequence.
"min function" on page 167	The fn:min function returns the minimum of the values in a sequence.
"number function" on page 178	The fn:number function converts a value to the xs:double data type.
"round function" on page 184	The fn:round function returns the integer that is closest to a specific numeric value.
"round-half-to-even function" on page 185	The fn:round-half-to-even function returns the numeric value with a specified precision that is closest to a specific numeric value.
"sum function" on page 198	The fn:sum function returns the sum of the values in a sequence.

Date, time, and duration functions

Function	Description
"adjust-date-to-timezone function" on page 133	The fn:adjust-date-to-timezone function adjusts an xs:date value for a specific time zone or removes the time zone component from the value.
"adjust-dateTime-to-timezone function" on page 135	The fn:adjust-dateTime-to-timezone function adjusts an xs:dateTime value for a specific time zone or removes the time zone component from the value.

Function	Description
"adjust-time-to-timezone function" on page 137	The fn:adjust-time-to-timezone function adjusts an xs:time value for a specific time zone or removes the time zone component from the value.
"current-date function" on page 145	The fn:current-date function returns the current date in the implicit time zone of UTC.
"current-dateTime function" on page 146	The fn:current-dateTime function returns the current date and time in the implicit time zone of UTC.
"current-local-date function" on page 146	The db2-fn:current-local-date function returns the current date in the local time zone.
"current-local-dateTime function" on page 146	The db2-fn:current-local-dateTime function returns the current date and time in the local time zone.
"current-local-time function" on page 147	The db2-fn:current-local-time function returns the current time in the local time zone.
"current-time function" on page 147	The fn:current-time function returns the current time in the implicit time zone of UTC.
"dateTime function" on page 148	The fn:dateTime function constructs an xs:dateTime value from an xs:date value and an xs:time value.
"day-from-date function" on page 149	The fn:day-from-date function returns the day component of an xs:date value.
"day-from-dateTime function" on page 149	The fn:day-from-dateTime function returns the day component of an xs:dateTime value.
"days-from-duration function" on page 150	The fn:days-from-duration function returns the days component of a duration.
"hours-from-dateTime function" on page 157	The fn:hours-from-dateTime function returns the hours component of an xs:dateTime value.
"hours-from-duration function" on page 158	The fn:hours-from-duration function returns the hours component of a duration value.
"hours-from-time function" on page 159	The fn:hours-from-time function returns the hours component of an xs:time value.
"implicit-timezone function" on page 159	The fn:implicit-timezone function returns the implicit time zone value of PT0S, which is of type xs:dayTimeDuration. The value PT0S indicates that UTC is the implicit time zone.
"local-timezone function" on page 163	The db2-fn:local-timezone function returns the time zone of the local system.
"minutes-from-dateTime function" on page 168	The fn:minutes-from-dateTime function returns the minutes component of an xs:dateTime value.
"minutes-from-duration function" on page 169	The fn:minutes-from-duration function returns the minutes component of a duration.

Function	Description
"minutes-from-time function" on page 170	The fn:minutes-from-time function returns the minutes component of an xs:time value.
"month-from-date function" on page 170	The fn:month-from-date function returns the month component of a xs:date value.
"month-from-dateTime function" on page 171	The fn:month-from-dateTime function returns the month component of an xs:dateTime value.
"months-from-duration function" on page 171	The fn:months-from-duration function returns the months component of a duration value.
"seconds-from-dateTime function" on page 187	The fn:seconds-from-dateTime function returns the seconds component of an xs:dateTime value.
"seconds-from-duration function" on page 187	The fn:seconds-from-duration function returns the seconds component of a duration.
"seconds-from-time function" on page 188	The fn:seconds-from-time function returns the seconds component of an xs:time value.
"timezone-from-date function" on page 199	The fn:timezone-from-date function returns the time zone component of an xs:date value.
"timezone-from-dateTime function" on page 200	The fn:timezone-from-dateTime function returns the time zone component of an xs:dateTime value.
"timezone-from-time function" on page 200	The fn:timezone-from-time function returns the time zone component of an xs:time value.
"year-from-date function" on page 207	The fn:year-from-date function returns the year component of an xs:date value.
"year-from-dateTime function" on page 207	The fn:year-from-dateTime function returns the year component of an xs:dateTime value.
"years-from-duration function" on page 208	The fn:years-from-duration function returns the years component of a duration.

Sequence functions

Function	Description
"count function" on page 145	The fn:count function returns the number of values in a sequence.
"data function" on page 147	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 151	The fn:deep-equal function compares two sequences to determine whether they meet the requirements for deep equality.
"distinct-values function" on page 153	The fn:distinct-values function returns the distinct values in a sequence.
"empty function" on page 154	The fn:empty function indicates whether an argument is an empty sequence.

Function	Description
"exactly-one function" on page 155	The fn:exactly-one function returns its argument if the argument contains exactly one item.
"exists function" on page 155	The fn:exists function can check for the existence of many different types of items, such as elements, attributes, text nodes, atomic values (for example, an integer) or XML documents.
"last function" on page 162	The fn:last function returns the number of values in the sequence that is being processed.
"index-of function" on page 160	The fn:index-of function returns the positions where an item appears in a sequence.
"insert-before function" on page 161	The fn:insert-before function inserts a sequence before a specific position in another sequence.
"one-or-more function" on page 178	The fn:one-or-more function returns its argument if the argument contains one or more items.
"position function" on page 179	The fn:position function returns the position of the context item in the sequence that is being processed.
"remove function" on page 180	The fn:remove function removes an item from a sequence.
"reverse function" on page 183	The fn:reverse function reverses the order of the items in a sequence.
"subsequence function" on page 195	The fn:subsequence function returns a subsequence of a sequence.
"unordered function" on page 204	The fn:unordered function returns the items in a sequence in non-deterministic order.
"zero-or-one function" on page 208	The fn:zero-or-one function returns its argument if the argument contains one item or is the empty sequence.

QName functions

Function	Description
"in-scope-prefixes function" on page 160	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 163	The fn:local-name-from-QName function returns the local part of an xs:QName value.
"namespace-uri-for-prefix function" on page 174	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 175	The fn:namespace-uri-from-QName function returns the namespace URI part of an xs:QName value.

Function	Description
"QName function" on page 179	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 182	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 162	The fn:local-name function returns the local name property of a node.
"name function" on page 172	The fn:name function returns the prefix and local name parts of a node name.
"namespace-uri function" on page 173	The fn:namespace-uri function returns the namespace URI of the qualified name for a node.
"node-name function" on page 175	The fn:node-name function returns the expanded QName of a node.
"root function" on page 184	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 152	The fn:default-collation function returns a URI that represents the default collation that is defined for the database.
"sqlquery function" on page 189	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 205	The db2-fn:xmlcolumn function retrieves a sequence from a column in the currently connected DB2 database.

adjust-date-to-timezone function

The fn:adjust-date-to-timezone function adjusts an xs:date value for a specific time zone or removes the time zone component from the value.

Syntax

```
►►──fn:adjust-date-to-timezone(date-value──────────────────)─────────────────►◄
                               └─,timezone-value─┘
```

date-value
> The date value that is to be adjusted.
>
> *date-value* is of type xs:date, or is an empty sequence.

timezone-value
> A duration that represents the time zone to which *date-value* is to be adjusted.
>
> *timezone-value* can be an empty sequence or a single value of type xdt:dayTimeDuration between -PT14H and PT14H, inclusive. The value can have an integer number of minutes and must not have a seconds component. If *timezone-value* is not specified, the default value is PT0H, which represents UTC.

Returned value

The returned value is either a value of type xs:date or an empty sequence depending on the parameters that are specified. If *date-value* is not an empty sequence, the returned value is of type xs:date. The following table describes the possible returned values:

Table 33. Types of input values and returned value for fn:adjust-date-to-timezone

date-value	*timezone-value*	**Returned value**
date-value that contains a timezone component	An explicit value, or no value specified (duration of PT0H)	The *date-value* adjusted for the time zone represented by *timezone-value*.
date-value that contains a timezone component	An empty sequence	The *date-value* with no timezone component.
date-value that does not contain a timezone component	An explicit value, or no value specified (duration of PT0H)	The *date-value* with a timezone component. The timezone component is the time zone represented by *timezone-value*. The date component is not adjusted for the time zone.
date-value that does not contain a timezone component	An empty sequence	The *date-value*.
An empty sequence	An explicit value, empty sequence, or no value specified	An empty sequence.

When adjusting *date-value* to a different time zone, *date-value* is treated as a dateTime value with time component 00:00:00. The returned value contains the timezone component represented by *timezone-value*. The following function calculates the adjusted date value:

```
xs:date(fn:adjust-dateTime-to-timezone(xs:dateTime(date-value),timezone-value))
```

Examples

In the following examples, the variable $tz is a duration of -10 hours, defined as `xdt:dayTimeDuration("-PT10H")`.

The following function adjusts the date value for May 7, 2002 in the UTC+1 time zone. The function specifies a *timezone-value* of -PT10H.

```
fn:adjust-date-to-timezone(xs:date("2002-05-07+01:00"), $tz)
```

The returned date value is 2002-05-06-10:00. The date is adjusted to the UTC-10 time zone.

The following function adds a timezone component to the date value for March 7, 2002 without a timezone component. The function specifies a *timezone-value* of -PT10H.

```
fn:adjust-date-to-timezone(xs:date("2002-03-07"), $tz)
```

The returned value is 2002-03-07-10:00. The timezone component is added to the date value.

The following function adjusts the date value for February 9, 2002 in the UTC-7 time zone. Without a *timezone-value* specified, the function uses the default *timezone-value* PT0H.

```
fn:adjust-date-to-timezone(xs:date("2002-02-09-07:00"))
```

The returned date is 2002-02-09Z, the date is adjusted to UTC.

The following function removes the timezone component from the date value for May 7, 2002 in the UTC-7 time zone. The *timezone-value* is an empty sequence.

```
fn:adjust-date-to-timezone(xs:date("2002-05-07-07:00"), ())
```

The returned value is 2002-05-07.

adjust-dateTime-to-timezone function

The fn:adjust-dateTime-to-timezone function adjusts an xs:dateTime value for a specific time zone or removes the time zone component from the value.

Syntax

```
►►──fn:adjust-dateTime-to-timezone(dateTime-value─────────────────)──────────►◄
                                                 └─,timezone-value─┘
```

dateTime-value
> The dateTime value that is to be adjusted.

> *dateTime-value* is of type xs:dateTime, or is an empty sequence.

timezone-value
> A duration that represents the time zone to which *dateTime-value* is to be adjusted.

> *timezone-value* can be an empty sequence or a single value of type xdt:dayTimeDuration between -PT14H and PT14H, inclusive. The value can have an integer number of minutes and must not have a seconds component. If *timezone-value* is not specified, the default value is PT0H, which represents UTC.

Returned value

The returned value is either a value of type xs:dateTime or is an empty sequence depending on the types of input values. If *dateTime-value* is not an empty sequence, the returned value is of type xs:dateTime. The following table describes the possible returned values:

Table 34. Types of input values and returned value for fn:adjust-dateTime-to-timezone

dateTime-value	*timezone-value*	**Returned value**
dateTime-value that contains a timezone component	An explicit value, or no value specified (duration of PT0H)	The *dateTime-value* adjusted to the time zone represented by *timezone-value*. The returned value contains the timezone component represented by *timezone-value*.
dateTime-value that contains a timezone component	An empty sequence	The *dateTime-value* with no timezone component.
dateTime-value that does not contain a timezone component	An explicit value, or no value specified (duration of PT0H)	The *dateTime-value* with a timezone component. The timezone component is the time zone represented by *timezone-value*. The date and time components are not adjusted to the time zone.
dateTime-value that does not contain a timezone component	An empty sequence	The *dateTime-value*.
An empty sequence	An explicit value, empty sequence, or no value specified	An empty sequence.

Examples

In the following examples, the variable $tz is a duration of -10 hours, defined as `xdt:dayTimeDuration("-PT10H")`.

The following function adjusts the dateTime of March 7, 2002 at 10 am in the UTC-7 time zone to the timezone specified by *timezone-value* of -PT10H.
```
fn:adjust-dateTime-to-timezone(xs:dateTime("2002-03-07T10:00:00-07:00"), $tz)
```

The returned dateTime is 2002-03-07T07:00:00-10:00.

The following function adjusts the dateTime value for March 7, 2002 at 10 am. The *dateTime-value* does not have a timezone component, and the function specifies a *timezone-value* of -PT10H.
```
fn:adjust-dateTime-to-timezone(xs:dateTime("2002-03-07T10:00:00"), $tz)
```

The returned dateTime is 2002-03-07T10:00:00-10:00.

In the following function adjusts the dateTime value for June 4, 2006 at 10 am in the UTC-7 time zone. Without a *timezone-value* specified, the function uses the default timezone value of PT0H.
```
fn:adjust-dateTime-to-timezone(xs:dateTime("2006-06-04T10:00:00-07:00"))
```

The returned dateTime is 2006-06-04T17:00:00Z, the dateTime adjusted to UTC.

The following function removes the timezone component from the dateTime value for March 7, 2002 at 10 am in the UTC-7 time zone. The *timezone-value* value is the empty sequence.

```
fn:adjust-dateTime-to-timezone(xs:dateTime("2002-03-07T10:00:00-07:00"), ())
```

The returned dateTime is 2002-03-07T10:00:00.

adjust-time-to-timezone function

The fn:adjust-time-to-timezone function adjusts an xs:time value for a specific time zone or removes the time zone component from the value.

Syntax

```
►►──fn:adjust-time-to-timezone(time-value──┬──────────────┬──)──────────►◄
                                           └─,timezone-value─┘
```

time-value
> The time value that is to be adjusted.
>
> *time-value* is of type xs:time, or is an empty sequence.

timezone-value
> A duration that represents the time zone to which *time-value* is to be adjusted.
>
> *timezone-value* can be an empty sequence or a single value of type xdt:dayTimeDuration between -PT14H and PT14H, inclusive. The value can have an integer number of minutes and must not have a seconds component. If *timezone-value* is not specified, the default value is PT0H, which represents UTC.

Returned value

The returned value is either a value of type xs:time or an empty sequence depending on the parameters that are specified. If *time-value* is not an empty sequence, the returned value is of type xs:time. The following table describes the possible returned values:

Table 35. Types of input values and returned value for fn:adjust-time-to-timezone

date-value	timezone-value	Returned value
time-value that contains a timezone component	An explicit value, or no value specified (duration of PT0H)	The *time-value* adjusted for the time zone represented by *timezone-value*. The returned value contains the timezone component represented by *timezone-value*. If the time zone adjustment crosses over midnight, the change in date is ignored.
time-value that contains a timezone component	An empty sequence	The *time-value* with no timezone component.

Table 35. Types of input values and returned value for fn:adjust-time-to-timezone (continued)

date-value	timezone-value	Returned value
time-value that does not contain a timezone component	An explicit value, or no value specified (duration of PT0H)	The *time-value* with a timezone component. The timezone component is the timezone represented by *timezone-value*. The time component is not adjusted for the time zone.
time-value that does not contain a timezone component	An empty sequence	The *time-value*.
An empty sequence	An explicit value, empty sequence, or no value specified	An empty sequence.

Examples

In the following examples, the variable $tz is a duration of -10 hours, defined as `xdt:dayTimeDuration("-PT10H")`.

The following function adjusts the time value for 10:00 am in the UTC-7 time zone, and the function specifies a *timezone-value* of -PT10H.

`fn:adjust-time-to-timezone(xs:time("10:00:00-07:00"), $tz)`

The returned value is 7:00:00-10:00. The time is adjusted to the time zone represented by the duration -PT10H.

The following function adjusts the time value for 1:00 pm. The time value does not have a timezone component.

`fn:adjust-time-to-timezone(xs:time("13:00:00"), $tz)`

The returned value is 13:00:00-10:00. The time contains a timezone component represented by the duration -PT10H.

The following function adjusts the time value for 10:00 am in the UTC-7 time zone. The function does not specify a *timezone-value* and uses the default value of PT0H.

`fn:adjust-time-to-timezone(xs:time("10:00:00-07:00"))`

The returned value is 17:00:00Z, the time adjusted to UTC.

The following function removes the timezone component from the time value 8:00 am in the UTC-7 time zone. The *timezone-value* is the empty sequence.

`fn:adjust-time-to-timezone(xs:time("08:00:00-07:00"), ())`

The returned value is 8:00:00.

The following example compares two times. The time zone adjustment crosses over the midnight and cause a date change. However, fn:adjust-time-to-timezone ignores date changes.

```
fn:adjust-time-to-timezone(xs:time("01:00:00+14:00"), $tz)
   eq xs:time("01:00:00-10:00")
```

The returned value is true.

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.

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".

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 (EBV) depends on the value of *sequence-expression*:

Table 36. EBVs returned for specific types of values in XQuery

Description of value	EBV returned
An empty sequence	false

Table 36. EBVs returned for specific types of values in XQuery (continued)

Description of value	EBV returned
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 if 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.

ceiling function

The fn:ceiling function returns the smallest integer that is greater than or equal to a specific 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.

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'.

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─┘
```

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'.

contains function

The fn:contains function determines whether a string contains a specific substring. The search string is matched using the default collation.

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.

Limitation of length

> The length of *substring* is limited to 32000 bytes.

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.

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.

current-date function

The fn:current-date function returns the current date in the implicit time zone 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.

current-dateTime function

The fn:current-dateTime function returns the current date and time in the implicit time zone 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.

current-local-date function

The db2-fn:current-local-date function returns the current date in the local time zone.

Syntax

►►──db2-fn:current-local-date()──►◄

Returned value

The returned value is an xs:date value that is the current date. The returned value does not include a time zone component.

For example, if you invoke this function on 2 December, 2009 at 3:00 Greenwich Mean Time (GMT) and the local time zone is Eastern Standard Time (-PT5H), the returned value is 2009-12-01.

current-local-dateTime function

The db2-fn:current-local-dateTime function returns the current date and time in the local time zone.

Syntax

►►──db2-fn:current-local-dateTime()────────────────────────────────────►◄

Returned value

The returned value is an xs:dateTime value that is the current date and time. The returned value does not include a time zone component.

For example, if you invoke this function on 2 December 2009 at 6:25 in Toronto (timezone -PT5H), an example of a returned value is `2009-12-02T06:25:30.864001`.

current-local-time function

The db2-fn:current-local-time function returns the current time in the local time zone.

Syntax

```
►►──db2-fn:current-local-time()──────────────────────────────────►◄
```

Returned value

The returned value is an xs:time value that is the current time. The returned value does not include a time zone component.

For example, if you invoke the function at 6:31 Greenwich Mean Time (GMT), and the local time zone is Eastern Standard Time (-PT5H), an example returned value is `01:31:35.519001`.

current-time function

The fn:current-time function returns the current time in the implicit time zone 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.

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).

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.

day-from-date function

The fn:day-from-date function returns the day component of an xs:date value.

Syntax

►►──fn:day-from-date(*date-value*)───►◄

date-value
>   The date value from which the day component is to be extracted.
>
>   *date-value* is of type xs:date, or is an empty sequence.

Returned value

If *date-value* is of type xs:date, the returned value is of type xs:integer, and the value is between 1 and 31, inclusive. The value is the day component of *date-value*.

If *date-value* is an empty sequence, the returned value is an empty sequence.

Example

The following function returns the day component of the date value for June 1, 2005.

```
fn:day-from-date(xs:date("2005-06-01"))
```

The returned value is 1.

day-from-dateTime function

The fn:day-from-dateTime function returns the day component of an xs:dateTime value.

Syntax

►►──fn:day-from-dateTime(*dateTime-value*)─────────────────────────────────►◄

dateTime-value
>   The dateTime value from which the day component is to be extracted.
>
>   *dateTime-value* is of type xs:dateTime, or is an empty sequence.

Returned value

If *dateTime-value* is of type xs:dateTime, the returned value is of type xs:integer, and the value is between 1 and 31, inclusive. The value is the day component of *dateTime-value*.

If *dateTime-value* is an empty sequence, the returned value is an empty sequence.

Example

The following function returns the day component of the dateTime value for January 31, 2005 at 8:00 pm in the UTC+4 time zone.

```
fn:day-from-dateTime(xs:dateTime("2005-01-31T20:00:00+04:00"))
```

The returned value is 31.

---

days-from-duration function

The fn:days-from-duration function returns the days component of a duration.

Syntax

▶▶──fn:days-from-duration(*duration-value*)────────────────◀◀

duration-value
> The duration value from which the days component is to be extracted.
>
> *duration-value* is an empty sequence, or is a value that has one of the following types: xdt:dayTimeDuration, xs:duration, or xdt:yearMonthDuration.

Returned value

The return value depends on the type of *duration-value*:
- If *duration-value* is of type xdt:dayTimeDuration or is of type xs:duration, the returned value is of type xs:integer, and is the days component of *duration-value* cast as xdt:dayTimeDuration. The returned value is negative if *duration-value* is negative.
- If *duration-value* is of type xdt:yearMonthDuration, the returned value is 0.
- If *duration-value* is an empty sequence, the returned value is an empty sequence.

The days component of *duration-value* cast as xdt:dayTimeDuration is the integer number of days computed as (S idiv 86400). The value S is the total number of seconds of *duration-value* cast as xdt:dayTimeDuration to remove the years and months components.

Examples

This function returns the days component of the duration -10 days and 0 hours.

```
fn:days-from-duration(xdt:dayTimeDuration("-P10DT00H"))
```

The returned value is -10.

This function returns the days component of the duration 3 days and 55 hours.

```
fn:days-from-duration(xdt:dayTimeDuration("P3DT55H"))
```

The returned value is 5. When calculating the total number of days in the duration, 55 hours is converted to 2 days and 7 hours. The duration is equal to P5D7H which has a days component of 5 days.

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 37. 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.

Table 37. Deep equality for nodes in a sequence (continued)

Node kind of both item-1 and item-2	Conditions for deep equality

Element	All of the following conditions must be true:
	
	• *item-1* and *item-2* have the same name, which means that their namespace URIs match and their local names match. Namespace prefixes are ignored. Name matching is done using a binary comparison.
	
	• *item-1* and *item-2* have the same number of attributes, and every attribute of *item-1* is deep-equal to an attribute of *item-2*.
	
	• One of the following conditions is true:
	
	  – 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 *item-1* is deep-equal to the sequence of the text and element children of *item-2*.
	
	  – Both nodes are validated with a simple type (such as xs:decimal) or a type that has simple content (such as a "temperature" type whose content is xs:decimal), and the typed value of *item-1* is deep-equal to the typed value of *item-2*.
	
	  – Both nodes are validated with a type that permits no content (neither text nor child elements).
	
	  – Both nodes are validated with a type that permits only child elements (no text), and each child element of *item-1* is deep-equal to the corresponding child element of *item-2*.

Attribute	All of the following conditions must be true: • *item-1* and *item-2* have the same name, which means that their namespace URIs match and their local names match. Namespace prefixes are ignored. Name matching is done using a binary comparison. • The typed value of *item-1* is deep-equal to the typed value of *item-2*.
Text	The content property values are equal when compared as strings with the eq operator using the default collation.
Comment	The content property values are equal when compared as strings with the eq operator using the default collation.
Processing instruction	All of the following conditions must be true: • *item-1* and *item-2* have the same name. • The content property values are equal when compared as strings with the eq operator using the default collation.

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.

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 specifying CLDR181_LEN as the collation. When querying this database with the fn:default-collation function, the following value is returned:

```
http://www.ibm.com/xmlns/prod/db2/sql/collations?name=CLDR181_LEN_AN_CX_EX_FX_HX_NX_S3
```

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 using the default collation. 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 can be (1, 'a', 'A', 'Hello') or (1.0, 'A', 'a', 'Hello').

empty function

The fn:empty function indicates whether an 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.

ends-with function

The fn:ends-with function determines whether a string ends with a specific substring. The search string is matched using the default collation.

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.

Limitation of length

  The length of *substring* is limited to 32000 bytes.

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.

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.

exists function

The fn:exists function can check for the existence of many different types of items, such as elements, attributes, text nodes, atomic values (for example, an integer) or XML documents.

If the XQuery expression specified as its argument, *sequence-expression*, produces an empty result (the empty sequence), then fn:exists returns **false**. If the argument returns anything but the empty sequence, then fn:exists returns **true**.

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. If *sequence-expression* produces the empty sequence, the returned value is false.

Examples

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.

The next example checks whether there is an element, customer, with a child element, phone. If there is, the fn:exists function returns true:

```
fn:exists($info/customer/phone)
```

The following example returns true if there is an element, customer, which has an attribute, Cid:

```
fn:exists($info/customer/@Cid)
```

The next example checks whether the element, comment, has a text node. In this example, if the comment element is an empty element it has no text node, so fn:exists returns false. Also, if there is no comment element at all, fn:exists returns false:

```
fn:exists($info/customer/comment/text())
```

The final example checks whether there is any XML document in the XML column INFO of the CUSTOMER table:

```
fn:exists( db2-fn:xmlcolumn("CUSTOMER.INFO") )
```

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.

floor function

The fn:floor function returns the largest integer that is less than or equal to a specific 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.

hours-from-dateTime function

The fn:hours-from-dateTime function returns the hours component of an xs:dateTime value.

Syntax

▶▶──fn:hours-from-dateTime(*dateTime-value*)────────────────────◀◀

dateTime-value
> The dateTime value from which the hours component is to be extracted.
>
> *dateTime-value* is of type xs:dateTime, or is an empty sequence.

Returned value

If *dateTime-value* is of type xs:dateTime, the returned value is of type xs:integer, and the value is between 0 and 23, inclusive. The value is the hours component of *dateTime-value*.

If *dateTime-value* is an empty sequence, the returned value is an empty sequence.

Example

The following function returns the hours component of the dateTime value for January 31, 2005 at 2:00 pm in the UTC-8 time zone.

```
fn:hours-from-dateTime(xs:dateTime("2005-01-31T14:00:00-08:00"))
```

The returned value is 14.

hours-from-duration function

The fn:hours-from-duration function returns the hours component of a duration value.

Syntax

▶▶──fn:hours-from-duration(*duration-value*)────────────────────◀◀

duration-value
> The duration value from which the hours component is to be extracted.
>
> *duration-value* is an empty sequence or is a value that has one of the following types: xdt:dayTimeDuration, xs:duration, or xdt:yearMonthDuration.

Returned value

The return value depends on the type of *duration-value*:

- If *duration-value* is of type xdt:dayTimeDuration or is of type xs:duration, the returned value is of type xs:integer, and is a value between -23 and 23, inclusive. The value is the hours component of *duration-value* cast as xdt:dayTimeDuration. The value is negative if *duration-value* is negative.
- If *duration-value* is of type xdt:yearMonthDuration, the returned value is of type xs:integer and is 0.
- If *duration-value* is an empty sequence, the returned value is an empty sequence.

The hours component of *duration-value* cast as xdt:dayTimeDuration is the integer number of hours computed as ((*S* mod 86400) idiv 3600). The value *S* is the total number of seconds of *duration-value* cast as xdt:dayTimeDuration to remove the

days and months component. The value 86400 is the number of seconds in a day, and 3600 is the number of seconds in an hour.

Example

The following function returns the hours component of the duration 126 hours.

```
fn:hours-from-duration(xdt:dayTimeDuration("PT126H"))
```

The returned value is 6. When calculating the total number of hours in the duration, 126 hours is converted to 5 days and 6 hours. The duration is equal to P5DT6H which has an hours component of 6 hours.

hours-from-time function

The fn:hours-from-time function returns the hours component of an xs:time value.

Syntax

```
►►──fn:hours-from-time(time-value)─────────────────────────────────────────►◄
```

time-value
> The time value from which the hours component is to be extracted.
>
> *time-value* is of type xs:time, or is an empty sequence.

Returned value

If *time-value* is not an empty sequence, the returned value is of type xs:integer, and the value is between 0 and 23, inclusive. The value is the hours component of *time-value*.

If *time-value* is an empty sequence, the returned value is an empty sequence.

Example

The following function returns the hours component of the time value for 9:30 am in the UTC-8 time zone.

```
fn:hours-from-time(xs:time("09:30:00-08:00"))
```

The returned value is 9.

implicit-timezone function

The fn:implicit-timezone function returns the implicit time zone value of PT0S, which is of type xs:dayTimeDuration. The value PT0S indicates that UTC is the implicit time zone.

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()
```

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.mycompany.com";
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.

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 using the default collation. 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).

insert-before function

The fn:insert-before function inserts a sequence before a specific 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).

last function

The fn:last function returns the number of values in the sequence that is 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

The following example uses the function as a predicate expression to return last item in the current sequence:

```
(<a/>, <b/>, <c/>)[fn:last()]
```

The returned value is <c/>.

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*.

Example

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.

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".

local-timezone function

The db2-fn:local-timezone function returns the time zone of the local system.

Syntax

```
▶▶──db2-fn:local-timezone()────────────────────────────────────────────◀◀
```

Returned value

The returned value is an xdt:dayTimeDuration value that represents the local time zone offset from Coordinated Universal Time (UTC).

For example, if you invoke this function in the local time zone of Eastern Standard Time, the returned value is -PT5H.

lower-case function

The fn:lower-case function converts a string to lowercase.

Syntax

```
▶▶──fn:lower-case(source-string──┬──────────────────┬──)──────────────────◀◀
                                 └─,──locale-name──┘
```

source-string
> The string that is to be converted to lowercase.
>
> *source-string* is of type xs:string, or is the empty sequence.

locale-name
> A string containing the locale to be used for the lowercase operation.
>
> *locale-name* is of type xs:string, or is the empty sequence. If *locale-name* is not the empty sequence, the value of *locale-name* is not case sensitive and must be a valid locale or a string of length zero.

Returned value

If *source-string* is not the empty sequence, the returned value is *source-string* with each character converted to its lowercase correspondent. If *locale-name* is not specified, is the empty sequence, or is a string of length zero, then the lowercase rules as defined in the Unicode standard are used. Otherwise, the lowercase rules for the specified locale are used. 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 zero.

Examples

The following function converts the string "Wireless Router TB2561" to lowercase:

```
fn:lower-case("Wireless Router TB2561")
```

The returned value is "wireless router tb2561"

The following function specifies the Turkish locale tr_TR and converts the letter "I" and the numeric character reference İ (the character reference for Latin upper case I with dot above).

```
fn:lower-case("I&#x130;", "tr_TR")
```

The returned value consists of two characters, the character represented by ı (Latin small letter dotless i), and the letter "i." For the Turkish locale, the letter "I" is converted to the character represented by ı (Latin small letter dotless i), and İ (Latin upper case I with dot above) is converted to "i."

The following function does not specify a locale and converts the letter "I" to lowercase using the rules defined in the Unicode standard.

```
fn:lower-case("I")
```

The returned value is the letter "i."

matches function

The fn:matches function determines whether a string matches a specific 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 the 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.

Limitation of length

The length of *source-string* and *pattern* is limited to 32000 bytes.

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 "abbcacadbdcd".

```
fn:matches("abbcacadbdcd","(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.

Example of ignoring spaces and capitalization when matching a pattern: The following function uses the i and x flags to ignore capitalization and spaces when determining whether the string "abc1234" matches the pattern "ABC 1234."

```
fn:matches("abc1234","ABC 1234", "ix")
```

The returned value is true.

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
- 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 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.

String values are compared using the default collation.

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".

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.

String values are compared using the default collation.

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".

minutes-from-dateTime function

The fn:minutes-from-dateTime function returns the minutes component of an xs:dateTime value.

Syntax

►►──fn:minutes-from-dateTime(*dateTime-value*)──────────────────────►◄

dateTime-value
> The dateTime value from which the minutes component is to be extracted.
>
> *dateTime-value* is of type xs:dateTime, or is an empty sequence.

Returned value

If *dateTime-value* is of type xs:dateTime, the returned value is of type xs:integer, and the value is between 0 and 59, inclusive. The value is the minutes component of *dateTime-value*.

If *dateTime-value* is an empty sequence, the returned value is an empty sequence.

Example

The following function returns the minutes component from the dateTime value for January 23, 2005 at 9:42 am in the UTC-8 time zone.

```
fn:minutes-from-dateTime(xs:dateTime("2005-01-23T09:42:00-08:00"))
```

The returned value is 42.

minutes-from-duration function

The fn:minutes-from-duration function returns the minutes component of a duration.

Syntax

▶▶──fn:minutes-from-duration(*duration-value*)──────────────────────◀◀

duration-value
> The duration value from which the minutes component is to be extracted.
>
> *duration-value* is an empty sequence, or is a value that has one of the following types: xdt:dayTimeDuration, xs:duration, or xdt:yearMonthDuration.

Returned value

The return value depends on the type of *duration-value*:

- If *duration-value* is of type xdt:dayTimeDuration or is of type xs:duration, the returned value is of type xs:integer and is a value between -59 and 59, inclusive. The value is the minutes component of *duration-value* cast as xdt:dayTimeDuration. The value is negative if *duration-value* is negative.
- If *duration-value* is of type xdt:yearMonthDuration, the returned value is 0.
- If *duration-value* is an empty sequence, the returned value is an empty sequence.

The minutes component of *duration-value* cast as xdt:dayTimeDuration is the integer number of minutes computed as `((S mod 3600) idiv 60)`. The value *S* is the total number of seconds of *duration-value* cast as xdt:dayTimeDuration to remove the years and months components.

Example

The following function returns the minutes component of the duration 2 days, 16 hours, and 93 minutes.

```
fn:minutes-from-duration(xdt:dayTimeDuration("P2DT16H93M"))
```

The returned value is 33. When calculating the total number of minutes in the duration, 93 minutes is converted to 1 hour and 33 minutes. The duration is equal to P2DT17H33M which has a minutes component of 33 minutes.

minutes-from-time function

The fn:minutes-from-time function returns the minutes component of an xs:time value.

Syntax

►►──fn:minutes-from-time(*time-value*)─────────────────────────────────◄◄

time-value
> The time value from which the minutes component is to be extracted.
>
> *time-value* is of type xs:time, or is an empty sequence.

Returned value

If *time-value* is of type xs:time, the returned value is of type xs:integer, and the value is between 0 and 59, inclusive. The value is the minutes component of *time-value*.

If *time-value* is an empty sequence, the returned value is an empty sequence.

Example

The following function returns the minutes component of the time value for 8:59 am in the UTC-8 time zone.

```
fn:minutes-from-time(xs:time("08:59:00-08:00"))
```

The returned value is 59.

month-from-date function

The fn:month-from-date function returns the month component of a xs:date value.

Syntax

►►──fn:month-from-date(*date-value*)────────────────────────────────────◄◄

date-value
> The date value from which the month component is to be extracted.
>
> *date-value* is of type xs:date, or is an empty sequence.

Returned value

If *date-value* is of type xs:date, the returned value is of type xs:integer, and the value is between 1 and 12, inclusive. The value is the month component of *date-value*.

If *date-value* is an empty sequence, the returned value is an empty sequence.

Example

The following function returns the month component of the date value for December 1, 2005.

```
fn:month-from-date(xs:date("2005-12-01"))
```

The returned value is 12.

month-from-dateTime function

The fn:month-from-dateTime function returns the month component of an xs:dateTime value.

Syntax

►►──fn:month-from-dateTime(*dateTime-value*)──────────────────────────────►◄

dateTime-value
>   The dateTime value from which the month component is to be extracted.
>
>   *dateTime-value* is of type xs:dateTime, or is an empty sequence.

Returned value

If *dateTime-value* is of type xs:dateTime, the returned value is of type xs:integer, and the value is between 1 and 12, inclusive. The value is the month component of *dateTime-value*.

If *dateTime-value* is an empty sequence, the returned value is an empty sequence.

Example

The following function returns the month component of the dateTime value for October 31, 2005 at 8:15 am in the UTC-8 time zone.

```
fn:month-from-dateTime(xs:dateTime("2005-10-31T08:15:00-08:00"))
```

The returned value is 10.

months-from-duration function

The fn:months-from-duration function returns the months component of a duration value.

Syntax

►►──fn:months-from-duration(*duration-value*)─────────────────────────────►◄

duration-value
>   The duration value from which the months component is to be extracted.

>   *duration-value* is an empty sequence, or is a value that has one of the
>   following types: xdt:dayTimeDuration, xs:duration, or
>   xdt:yearMonthDuration.

Returned value

The return value depends on the type of *duration-value*:

- If *duration-value* is of type xs:duration or is of type xdt:yearMonthDuration, the
  returned value is of type xs:integer, and is a value is between -11 and 11,
  inclusive. The value is the months component of *duration-value* cast as
  xdt:yearMonthDuration. The value is negative if *duration-value* is negative.
- If *duration-value* is of type xdt:dayTimeDuration, the returned value is 0.
- If *duration-value* is an empty sequence, the returned value is an empty sequence.

The months component of *duration-value* cast as xdt:yearMonthDuration is the
integer number of months remaining from the total number of months of
duration-value divided by 12.

Examples

The following function returns the months component of the duration 20 years and
5 months.

```
fn:months-from-duration(xs:duration("P20Y5M"))
```

The returned value is 5.

The following function returns the months component of the yearMonthDuration
-9 years and -13 months.

```
fn:months-from-duration(xdt:yearMonthDuration("-P9Y13M"))
```

The returned value is -1. When calculating the total number of months in the
duration, -13 months is converted to -1 year and -1 month. The duration is equal to
-P10Y1M which has a month component of -1 month.

The following function returns the months component of the duration 14 years, 11
months, 40 days, and 13 hours.

```
xquery fn:months-from-duration(xs:duration("P14Y11M40DT13H"))
```

The returned value is 11.

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.mycompany.com";
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.mycompany.com";
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()
```

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:

Chapter 5. Built-in functions　**173**

- – *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.mycompany.com";
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.mycompany.com";
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()
```

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.mycompany.com";
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)
```

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"))
```

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.mycompany.com";
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)
```

normalize-space function

The fn:normalize-space function strips leading and trailing white space characters from a string and replaces each internal sequence of white space 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".

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.

Examples

The following function performs Unicode Normalization Form C on the string "ṃ" (a Latin lowercase letter m with a dot below):

```
fn:normalize-unicode("&#x6d;&#x323;","NFC")
```

The returned value is the UTF-8 character represented by the numeric character reference &x1e43;, a Latin lowercase letter m with a dot below.

The following example converts the normalized Unicode to the decimal codepoint:

```
fn:string-to-codepoints(fn:normalize-unicode("&#x6d;&#x323;", "NFC"))
```

The returned value is 7747.

not function

The fn:not function returns false if the effective boolean value of a sequence is true and returns 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 />)
```

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.

Examples

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.

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.

position function

The fn:position function returns the position of the context item in the sequence that is 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.

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 xs:QName.

> *QName* data type 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".

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).

replace function

The fn:replace function compares each set of characters within a string to a specific pattern, and then it replaces the characters that match the pattern with another set of characters.

Syntax

►►──fn:replace(*source-string*,*pattern*,*replacement-string*──┬──────────┬──)──────────►◄
                                                              └─,*flags*─┘

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.

Limitation of length

> The length of *source-string*, *pattern* and *replacement-string* is limited to 32000 bytes.

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 "abbcacadbdcd" with "ba".

```
fn:replace("abbcacadbdcd","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 "abbcacadbdcd".

```
fn:replace("abbcacadbdcd","a(.)","$1$1")
```

The returned value is "bbbcccddbdcd".

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)
```

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).

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.

round function

The fn:round function returns the integer that is closest to a specific 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.

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 specific 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)
```

seconds-from-dateTime function

The fn:seconds-from-dateTime function returns the seconds component of an xs:dateTime value.

Syntax

►►──fn:seconds-from-dateTime(*dateTime-value*)─────────────────────────►◄

dateTime-value
    The dateTime value from which the seconds component is to be extracted.

    dateTime-value is of type xs:dateTime, or is an empty sequence.

Returned value

If *dateTime-value* is of type xs:dateTime, the returned value is of type xs:decimal, and the value is greater than or equal to 0 and less than 60. The value is the seconds and fractional seconds component of *dateTime-value*.

If *dateTime-value* is an empty sequence, the returned value is an empty sequence.

Examples

The following function returns the seconds component of dateTime value for February 8, 2005 at 2:16:23 pm in theUTC-8 time zone.

```
fn:seconds-from-dateTime(xs:dateTime("2005-02-08T14:16:23-08:00"))
```

The returned value is 23.

The following function returns the seconds component of dateTime value for June 23, 2005 at 9:16:20.43 am in the UTC time zone.

```
fn:seconds-from-dateTime(xs:dateTime("2005-06-23T09:16:23.43Z"))
```

The returned value is 20.43.

seconds-from-duration function

The fn:seconds-from-duration function returns the seconds component of a duration.

Syntax

►►──fn:seconds-from-duration(*duration-value*)─────────────────────────►◄

duration-value
    The duration value from which the seconds component is to be extracted.

duration-value is an empty sequence, or is a value that has one of the following types: xdt:dayTimeDuration, xs:duration, or xdt:yearMonthDuration.

Returned value

The return value depends on the type of *duration-value*:

- If *duration-value* is of type xdt:dayTimeDuration, or is of type xs:duration, the returned value is of type xs:decimal, and is a value greater than -60 and less than 60. The value is the seconds and fractional seconds component of *duration-value* cast as xdt:dayTimeDuration. The value is negative if *duration-value* is negative.
- If *duration-value* is of type xdt:yearMonthDuration, the returned value is of type xs:integer and is 0.
- If *duration-value* is an empty sequence, the returned value is an empty sequence.

The seconds and fractional seconds component of *duration-value* cast as xdt:dayTimeDuration is computed as (S mod 60). The value S is the total number of seconds and fractional seconds of *duration-value* cast as xdt:dayTimeDuration to remove the years and months components.

Example

The following function returns the seconds component of the duration 150.5 seconds.

```
fn:seconds-from-duration(xdt:dayTimeDuration("PT150.5S"))
```

The returned value is 30.5. When calculating the total number of seconds in the duration, 150.5 seconds is converted to 2 minutes and 30.5 seconds. The duration is equal to PT2M30.5S which has a seconds component of 30.5 seconds.

seconds-from-time function

The fn:seconds-from-time function returns the seconds component of an xs:time value.

Syntax

►►──fn:seconds-from-time(*time-value*)─────────────────────────────►◄

time-value
    The time value from which the seconds component is to be extracted.

    time-value is of type xs:time, or is an empty sequence.

Returned value

If *time-value* is of type xs:time, the returned value is of type xs:decimal, and the value is greater than or equal to zero and less than 60. The value is the seconds and fractional seconds component of *time-value*.

If *time-value* is an empty sequence, the returned value is an empty sequence.

Example

The following function returns the seconds component of the time value for 08:59:59 am in the UTC-8 time zone.

```
fn:seconds-from-time(xs:time("08:59:59-08:00"))
```

The returned value is 59.

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──┬──────────────────────────┬──)──────────────►◄
                                     │    ┌────────────────────┐ │
                                     └────┴─,─parameter-expression─┴─┘
```

string-literal

Contains a fullselect. The fullselect must specify a single-column result set, and the column must have the XML data type. The scope of the fullselect is a new SQL query scope, not a nested SQL query.

The fullselect cannot contain an isolation clause or lock-request clause.

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'''
```

The fullselect can contain calls to the PARAMETER function to reference the result value from each *parameter-expression* specified in the db2-fn:sqlquery function invocation. The PARAMETER function calls are substituted with the result value of the corresponding *parameter-expression* in the execution of the fullselect.

parameter-expression

An XQuery expression that returns a value. The result value of each *parameter-expression* can be referenced by a designated SQL function PARAMETER with an integer value argument in the SQL fullselect. The integer value is an index to the *parameter-expression* by its position in the db2-fn:sqlquery function invocation. The valid integer values are between 1 and the total number of the *parameter-expression* in the function invocation. For example, if the *string-literal* argument includes PARAMETER(1) and PARAMETER(2) in the SQL fullselect, the function invocation must specify

two XQuery *parameter-expression* arguments. `PARAMETER(1)` references the result of the first *parameter-expression* argument and `PARAMETER(2)` references the result of the second *parameter-expression* argument.

During the processing of the SQL fullselect, each PARAMETER function call is replaced with the result value of the corresponding *parameter-expression* in the db2-fn:sqlquery function invocation. Each *parameter-expression* is evaluated once regardless the number of times it is referenced in the SQL fullselect.

The result data type of the corresponding *parameter-expression* must be castable to the result type of the PARAMETER function according to the rules of XMLCAST. If not, an error is returned.

The result type of the PARAMETER function is determined as if it were a parameter marker in the SQL fullselect. For example, a parameter marker is indicated by a question mark (?), or a colon followed by a name (:*name*), in other contexts. If the result type cannot be determined for a PARAMETER function, an error is returned.

Tip: If an untyped parameter marker is not allowed in an operation, you can use the CAST specification or XMLCAST specification to specify a type. For example, to cast `PARAMETER(1)` to the type DOUBLE, use the following CAST specification, `CAST(PARAMETER(1) as double)`.

Returned value

The returned value is a sequence that is the result of the fullselect in *string-literal*. The fullselect is processed as an SQL statement, following the usual dynamic SQL rules for authorization and name resolution. If the fullselect contains any calls to the PARAMETER function, they are substituted with the result value of the XQuery expression of the corresponding *parameter-expression* argument when the fullselect is evaluated. 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 XML 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 XML 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'")
```

Example of fullselect using two PARAMETER function calls and one expression:
The following example returns the purchase ID, part ID, and the purchase date for
all the parts sold within the promotion dates.

```
xquery
for $po in db2-fn:xmlcolumn('PURCHASEORDER.PORDER')/PurchaseOrder,
  $item in $po/item/partid
for $p in db2-fn:sqlquery(
  "select description
  from product
  where promostart < parameter(1)
  and  promoend  >  parameter(1)",
  $po/@OrderDate )
where $p//@pid = $item
return
<RESULT>
  <PoNum>{data($po/@PoNum)}</PoNum>
  <PartID>{data($item)} </PartID>
  <PoDate>{data($po/@OrderDate)}</PoDate>
</RESULT>
```

During processing of the db2-fn:sqlquery function, both references to `parameter(1)`
return the value of the order date attribute $po/@OrderDate.

**Example of a fullselect using two PARAMETER function calls and two
expressions:** The following example uses the PURCHASEORDER table from the
DB2 SAMPLE database. The XQuery expression retrieves unshipped purchase
orders that have an order date before April 4, 2006, and lists the distinct part
numbers from each purchase order:

```
xquery
let $status := ( "Unshipped" ), $date := ( "2006-04-04" )
for $myorders in db2-fn:sqlquery(
  "select porder from purchaseorder
  where status = parameter(1)
  and orderdate < parameter(2)",
  $status, $date )
return
<LateOrder>
  <PoNum>
  {data($myorders/PurchaseOrder/@PoNum)}
  </PoNum>
  <PoDate>
  {data($myorders/PurchaseOrder/@OrderDate)}
  </PoDate>
  <Items>
    {for $itemID in distinct-values( $myorders/PurchaseOrder/item/partid )
    return
    <PartID>
    {$itemID}
    </PartID>}
  </Items>
</LateOrder>
```

During processing of the db2-fn:sqlquery function, the reference to `parameter(1)`
returns the result value of the expression `$status`, and the reference to
`parameter(2)` returns the result value of the expression `$date`.

When run against the SAMPLE database, the expression returns the following
result:

```
<LateOrder>
  <PoNum>5000</PoNum>
  <PoDate>2006-02-18</PoDate>
  <Items>
    <PartID>100-100-01</PartID>
    <PartID>100-103-01</PartID>
  </Items>
</LateOrder>
```

starts-with function

The fn:starts-with function determines whether a string begins with a specific
substring. The search string is matched using the default collation.

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.

Limitation of length

        The length of *substring* is limited to 32000 bytes.

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.

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
corresponds 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).

---

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<=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 (length of *source-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').

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'.

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 specific search string. The search string is matched using the default collation.

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.

Limitation of length

The length of *search-string* is limited to 32000 bytes.

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'.

substring-before function

The fn:substring-before function returns a substring that occurs in a string before the first occurrence of a specific search string. The search string is matched using the default collation.

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.

`Limitation of length`

The length of *search-string* is limited to 32000 bytes.

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 a string of length 0.
- 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'.

---

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".

timezone-from-date function

The fn:timezone-from-date function returns the time zone component of an xs:date value.

Syntax

▶▶──fn:timezone-from-date(*date-value*)────────────────────────◀◀

date-value
> The date value from which the timezone component is to be extracted.
>
> *date-value* is of type xs:date, or is an empty sequence.

Returned value

If *date-value* is of type xs:date and has an explicit timezone component, the returned value is of type xdt:dayTimeDuration, and the value is between -PT14H hours and PT14H, inclusive. The value is the deviation of the *date-value* timezone component from the UTC time zone.

If *date-value* does not have an explicit timezone component or is an empty sequence, the returned value is an empty sequence.

Example

The following function returns the timezone component of the date value for March 13, 2007 in the UTC-8 time zone.

```
fn:timezone-from-date(xs:date("2007-03-13-08:00"))
```

The returned value is -PT8H.

timezone-from-dateTime function

The fn:timezone-from-dateTime function returns the time zone component of an xs:dateTime value.

Syntax

►►──fn:timezone-from-dateTime(*dateTime-value*)──────────────────────────◄◄

dateTime-value
>   The dateTime value from which the timezone component is to be extracted.
>
>   *dateTime-value* is of type xs:dateTime, or is an empty sequence.

Returned value

If *dateTime-value* is of type xs:dateTime and has an explicit timezone component, the returned value is of type xdt:dayTimeDuration, and the value is between -PT14H and PT14H, inclusive. The value is the deviation of the *dateTime-value* timezone component from the UTC time zone.

If *dateTime-value* does not have an explicit timezone component, or is an empty sequence, the returned value is an empty sequence.

Examples

The following function returns the timezone component of the dateTime value for October 30, 2005 at 7:30 am in the UTC-8 time zone.

```
fn:timezone-from-dateTime(xs:dateTime("2005-10-30T07:30:00-08:00"))
```

The returned value is -PT8H.

The following function returns the timezone component of the dateTime value for January 1, 2005 at 2:30 pm in the UTC+10:30 time zone.

```
fn:timezone-from-dateTime(xs:dateTime("2005-01-01T14:30:00+10:30"))
```

The returned value is PT10H30M.

timezone-from-time function

The fn:timezone-from-time function returns the time zone component of an xs:time value.

Syntax

►►──fn:timezone-from-time(*time-value*)──────────────────────────────────◄◄

time-value
>   The time value from which the timezone component is to be extracted.
>
>   *time-value* is of type xs:time, or is an empty sequence.

Returned value

If *time-value* is of type xs:time and has an explicit timezone component, the returned value is of type xdt:dayTimeDuration, and the value is between -PT14H and PT14H, inclusive. The value is the deviation of the *time-value* timezone component from UTC time zone.

If *time-value* does not have an explicit timezone component, or is an empty sequence, the returned value is an empty sequence.

Examples

The following function returns the timezone component of the time value for 12 noon in the UTC-5 time zone.

```
fn:timezone-from-time(xs:time("12:00:00-05:00"))
```

The returned value is -PT5H.

In the following function, the time value for 1:00 pm does not have a timezone component.

```
fn:timezone-from-time(xs:time("13:00:00"))
```

The returned value is the empty sequence.

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.

Limitation of length

The length of *source-string* and *pattern* is limited to 32000 bytes.

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.").

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.

Limitation of length
>The length of *original-string* and *replacement-string* is limited to 32000 bytes.

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*. The characters are matched using a binary comparison.

  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'.

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.

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))
```

upper-case function

The fn:upper-case function converts a string to uppercase.

Syntax

```
►►──fn:upper-case(source-string──────────────)──────────────────◄◄
                              └,─locale-name─┘
```

source-string
   The string that is to be converted to uppercase.

   source-string has the xs:string data type, or is an empty sequence.

locale-name
> A string containing the locale to be used for the uppercase operation.
>
> *locale-name* is of type xs:string, or is the empty sequence. If *locale-name* is not the empty sequence, the value of *locale-name* is not case sensitive and must be a valid locale or a string of length zero.

Returned value

If *source-string* is not an empty sequence, the returned value is *source-string* with each character converted to its uppercase correspondent. If *locale-name* is not specified, is the empty sequence, or is a string of length zero, then the uppercase rules as defined in the Unicode standard are used. Otherwise, the uppercase rules for the specified locale are used. Every character that does not have an uppercase 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 zero.

Examples

The following function converts the string 'Test literal 1' to uppercase.

```
fn:upper-case('Test literal 1')
```

The returned value is 'TEST LITERAL 1'.

The following function specifies the Turkish locale tr_TR and converts the letter "i", and the numeric character reference ı (the character reference for Latin small letter dotless i).

```
fn:upper-case("i&#x131;", "tr_TR")
```

The returned value consists of two characters, the character represented by İ (Latin upper case I with dot above), and the letter "I". For the Turkish locale, the letter "i" is converted to character represented by İ (Latin upper case I with dot above), and character represented by ı (Latin small letter dotless i) is converted to the letter "I".

The following function does not specify a locale and converts two characters to uppercase using the rules defined in the Unicode standard.

```
fn:upper-case("&#x131;i")
```

The function returns the characters "II". fn:upper-case converts both the lowercase character ı and the letter "i" to the uppercase letter "I".

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 a DB2 database are in uppercase by default, the argument of db2-fn:xmlcolumn is usually in uppercase. The argument of the db2-fn:sqlquery function is processed by SQL, which automatically converts identifiers to uppercase.

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.DESCRIPTION')
```

Example that uses an implicit SQL schema: In the following example, the CURRENT SCHEMA special register in a DB2 database is set to SAMPLE, and so the function returns the same results as the previous example:

```
db2-fn:xmlcolumn('PRODUCT.DESCRIPTION')
```

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 lowercase 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 quotation marks):

  ```
  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"')
```

year-from-date function

The fn:year-from-date function returns the year component of an xs:date value.

Syntax

►►──fn:year-from-date(*date-value*)──────────────────────────────────────►◄

date-value
> The date value from which the year component is to be extracted.
>
> *date-value* is of type xs:date, or is an empty sequence.

Returned value

If *date-value* is of type xs:date, the returned value is of type xs:integer, The value is the year component of the *date-value*, a non-negative value.

If *date-value* is an empty sequence, the returned value is an empty sequence.

Example

The following function returns the year component of the date value for October 29, 2005.
```
fn:year-from-date(xs:date("2005-10-29"))
```

The returned value is 2005.

year-from-dateTime function

The fn:year-from-dateTime function returns the year component of an xs:dateTime value.

Syntax

►►──fn:year-from-dateTime(*dateTime-value*)──────────────────────────────►◄

dateTime-value
> The dateTime value from which the year component is to be extracted.
>
> *dateTime-value* is of type xs:dateTime, or is an empty sequence.

Returned value

If *dateTime-value* is of type xs:dateTime, the returned value is of type xs:integer. The value is the year component of the *dateTime-value*, a non-negative value.

If *dateTime-value* is an empty sequence, the returned value is an empty sequence.

Example

The following function returns the year component of the dateTime value for October 29, 2005 at 8:00 am in the UTC-8 time zone.

```
fn:year-from-dateTime(xs:dateTime("2005-10-29T08:00:00-08:00"))
```

The returned value is 2005.

years-from-duration function

The fn:years-from-duration function returns the years component of a duration.

Syntax

►►──fn:years-from-duration(*duration-value*)─────────────────────────────►◄

duration-value
> The duration value from which the years component is to be extracted.
>
> *duration-value* is an empty sequence, or is a value that has one of the following types: xdt:dayTimeDuration, xs:duration, or xdt:yearMonthDuration.

Returned value

The return value depends on the type of *duration-value*:

- If *duration-value* is of type xdt:yearMonthDuration or is of type xs:duration, the returned value is of type xs:integer. The value is the years component of *duration-value* cast as xdt:yearMonthDuration. The value is negative if *duration-value* is negative.
- If *duration-value* is of type xs:dayTimeDuration, the returned value is of type xs:integer and is 0.
- If *duration-value* is an empty sequence, the returned value is an empty sequence.

The years component of *duration-value* cast as xdt:yearMonthDuration is the integer number of years determined by the total number of months of *duration-value* cast as xdt:yearMonthDuration divided by 12.

Examples

The following function returns the years component of the duration -4 years, -11 months, and -320 days.

```
fn:years-from-duration(xs:duration("-P4Y11M320D"))
```

The returned value is -4.

The following function returns the years component of the duration 9 years and 13 months.

```
fn:years-from-duration(xdt:yearMonthDuration("P9Y13M"))
```

The returned value is 10. When calculating the total number of years in the duration, 13 months is converted to 1 year and 1 month. The duration is equal to P10Y1M which has a years component of 10 years.

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.

Chapter 6. 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 Recommendation *XQuery 1.0 and XPath 2.0 Functions and Operators*.

Syntax

RegularExpression
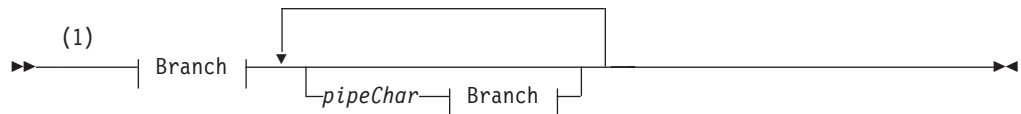
Branch:
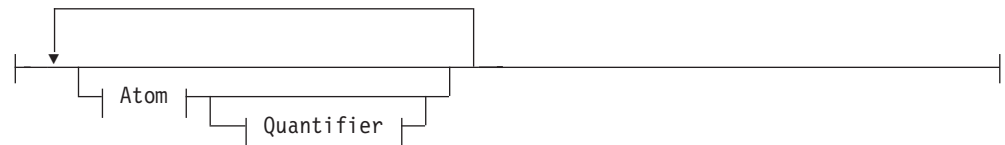
Atom:
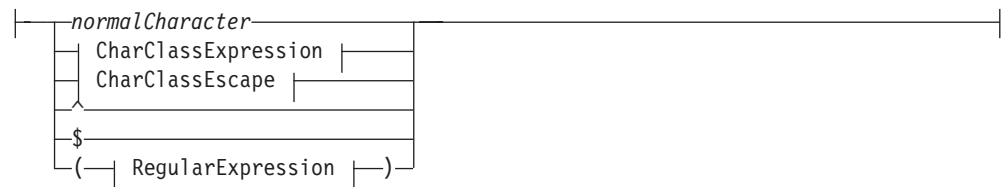
Quantifier:
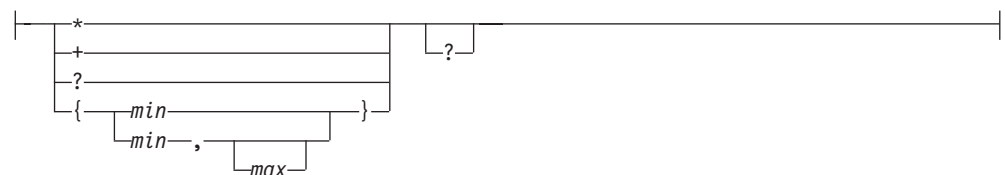
CharClassExpression:

CharGroup:

```
                  ┌──────────────────────────────────┐
                  ▼                                  │
┌─────┬──────────┬──XMLCharIncludeDash──────────────┼──────────────┤
      └─ ^ ─┘     ├──XMLChar────┬──dashChar──┬──XMLChar─────┐
                  └─charEscape──┘            └─charEscape──┘
                  └──CharClassEscape──────────┘
```

CharClassEscape:

```
├──┬──.──────────────────────┬──────────────────────────┤
   ├─charEscape──────────────┤
   ├─multiCharEscape─────────┤
   ├─\nonZeroDigit───────────┤
   ├─\p{IsblockName}─────────┤
   ├─\P{IsblockName}─────────┤
   ├─\p{charProperty}────────┤
   └─\P{charProperty}────────┘
```
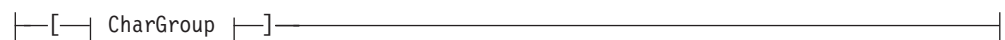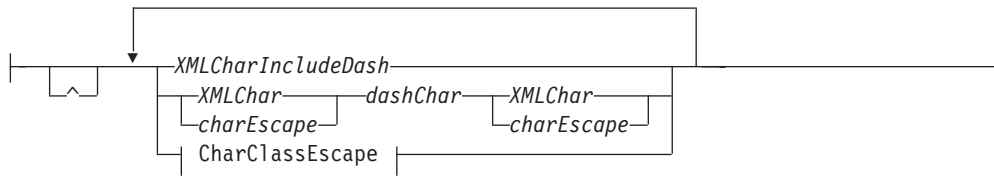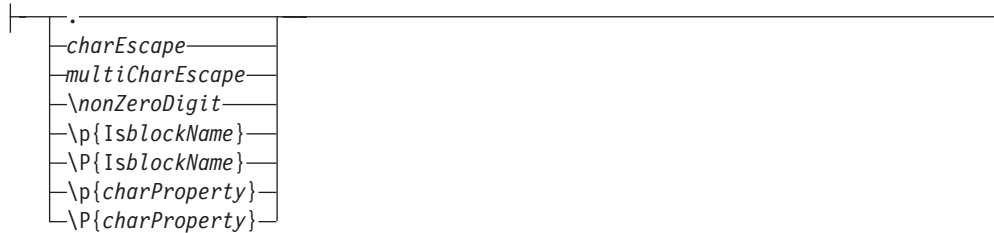
Notes:

1   The syntax for regular-expression represents the content of a string literal that
     cannot include whitespace characters other than as the specific meaning of
     the whitespace character as a pattern character. Do not consider spaces or
     portions between syntax elements as allowing any form of whitespace.

RegularExpression

A regular expression contains one or more branches. Branches are separated by
pipes (|), indicating that each branch is an alternative pattern.

pipeChar
    A pipe character (|) separates alternative branches in a regular expression.

Branch
    A branch consists of zero or more atoms, with each atom allowing an optional
    quantifier.

Atom

An atom is either a normal character, a character class expression, a character class
escape, or a parenthesized regular expression.

normalCharacter
    Any valid XML character that is not one of the metacharacters that is in
    Table 38 on page 214.

^   When used at the beginning of a branch, the caret (^) indicates that the pattern
     must match from the beginning of the string.

$ When used at the end of a branch, the dollar sign ($) indicates that the pattern
     must match from the end of the string.

Quantifier

The quantifier specifies the repetition of an atom in a regular expression. By
default, a quantifier will match as much as possible of the target string, using what
is referred to as a *greedy algorithm*. For example, the regular expression `'A.*A'`

matches the entire string 'ABACADA' because the substring between the required outer 'A' characters matches the requirement for any character any number of times. The default greedy algorithm can be changed by specifying the question mark (?) character after the quantifier. The question mark specifies that the pattern matching uses a *reluctant algorithm*, which matches to the next shortest substring from left to right in the target string that satisfies the regular expression. For example, the regular expression `A.*?A` matches the substrings 'ABA' and 'ADA' instead of matching the entire string 'ABACADA'. Characters of a substring that matches a regular expression by using the reluctant algorithm are not considered for further matches. This is why 'ACA' is not considered a match in the previous example. The reluctant algorithm is most useful with the fn:replace function because it processes matches and replacements from left to right.

For example, if you use the greedy algorithm in the function `fn:replace("nonsensical","n(.*)s","mus")` to replace the string of characters starting with "n" and ending with "s" with the string "mus", the returned value is 'musical'. The original string included substrings "nons" and "ns", which also matched the pattern scanning left to right for the next match, but the greedy algorithm did not operate on these matches because it found a longer enclosing match.

The result is different if you use the reluctant algorithm on the same string in the function `fn:replace("nonsensical","n(.*?)s","mus")`. The returned value is "musemusical". In this case, two replacements occurred within in the string. The first match replaced "nons" with "mus", and the second match replaced "ns" with "mus".

As another example, if you use the greedy algorithm to replace the character A that encloses any number of characters with the character X that encloses the same characters in the function `fn:replace("AbrAcAdAbrA","A(.*)A","X$1X")`, the returned value is "XbrAcAdAbrX". The original string included substrings "AbrA" and "AdA", which also matched the pattern when scanning left to right for the next match, but the greedy algorithm did not operate on these matches because it found a longer enclosing match.

The result is different if you use the reluctant algorithm on the same string in the function `fn:replace("AbrAcAdAbrA","A(.*?)A","X$1X")`. The returned value is "XbrXcXdXbrA". In this case, two replacements occurred within in the string: the first on "AbrA", and the second on "AdA". The final "A" in the string did not get replaced because the reluctant algorithm used all of the preceding "A" characters for other matches within the string. Other substrings that start and end with character "A", such as "AcA", "AcAdA", "AdAbrA" and "AbrA", within the original string are not considered because the reluctant algorithm considers the characters to be already used after they participate in a match to the pattern.

* Matches the atom zero or more times. Equivalent to the quantifier {0, }.

+ Matches the atom one or more times. Equivalent to the quantifier {1, }.

? Matches the atom zero or one times. Equivalent to the quantifier {0, 1}. When following another quantifier, indicates use of the reluctant algorithm instead of the greedy algorithm.

min
> Matches the atom at least *min* number of times. *min* must be a positive integer.
> - {*min*} matches the atom exactly *min* times.
> - {*min*, } matches the atom at least *min* times.

max

> Matches the atom at not more than *max* number of times. *max* must be a positive integer greater than or equal to *min*.
>
> - {0, *max*} matches the atom not more than *min* times.
> - {0, 0} matches only an empty string.

CharGroup

^
> Indicates the complement of the set of characters that are defined by the rest of the CharGroup.

dashChar
> The dash character (-)separates two characters that define the outer characters in a range of characters. A character range of the form s-e is the set of UCS2 code points that are greater than or equal to s and less than or equal to e such that:
>
> - s is not the backslash character (\)
> - If s is the first character in a CharGroup, it is not the caret character (^)
> - e is not the backslash character (\) or the opening bracket character ([)
> - The code point of e is greater than the code point of s

XMLCharIncludeDash
> A single character from the set of valid XML characters, excluding the backslash (\) and brackets ([]), but including the dash (-). The dash is valid as a character only at the beginning or the end of a CharGroup. The caret (^) at the beginning of a CharGroup indicates the complement of the group. Anywhere else in the group, the caret just matches the caret character. *XMLCharIncludeDash* can include any character that is matched by the regular expression [^\#5B#5D].

XMLChar
> A single character from the set of valid XML characters, excluding the backslash (\), brackets ([]), and the dash (-). The dash is valid as a character only at the beginning or the end of a CharGroup. The caret (^) at the beginning of a CharGroup indicates the complement of the group. Anywhere else in the group, the caret just matches the caret character. *XMLChar* can include any character that is matched by the regular expression [^\#2D#5B#5D].

charEscape
> A backslash followed by a single metacharacter, newline character, return character, or tab character. You must escape the characters that are in Table 38 in a regular expression to match them.

Table 38. Valid metacharacter escapes

Character escape	Character represented	Description
\n	#x0A	Newline
\r	#x0D	Return
\t	#x09	Tab
\\	\	Backslash
|			Pipe
\.	.	Period
\-	-	Dash
\^	^	Caret
\?	?	Question mark

Table 38. Valid metacharacter escapes (continued)

Character escape	Character represented	Description
\$	$	Dollar sign
*	*	Asterisk
\+	+	Plus sign
\{	{	Opening curly brace
\}	}	Closing curly brace
\(	(	Opening parenthesis
\)	)	Closing parenthesis
\[	[	Opening bracket
\]	]	Closing bracket

CharClassEscape

.
The period character (.) matches all characters except newline and return characters. The period character is quivalent to the expression [^\n\r].

\nonZeroDigit
Specifies a back reference that matches the string that was matched by a subexpression, which is surrounded by parentheses, in the *nonZeroDigit* position in the regular expression. *nonZeroDigit* must be between 1 and 9. The first 9 subexpressions can be referenced.

Note: For future upward compatibility, if a back reference is followed by a digit character, enclose the back reference in parentheses. For example, a back reference to the first subexpression that is followed by the digit 3 should be expressed as (/1)3 instead of /13 even though both currently produce the same result.

\P{**Is***blockName*}
Specifies the complement of a range of Unicode code points. The range is identified by *blockName*, as listed in *XML Schema Part 2: Datatypes Second Edition*.

\p{**Is***blockName*}
Specifies a character in a specific range of Unicode code points. The range is identified by *blockName*, as listed in *XML Schema Part 2: Datatypes Second Edition*.

charEscape
A backslash followed by a single metacharacter, newline character, return character, or tab character. You must escape the characters that are in Table 38 on page 214 in a regular expression to match them.

multiCharEscape
A backslash followed by a character that identifies commonly used sets of characters that are in Table 39 in a regular expression to match them.

Table 39. Multi-character escapes

Multi-character escape	Equivalent regular expression	Description
\s	[#x20\t\n\r]	Space, tab, newline, or return character.

Table 39. Multi-character escapes (continued)

Multi-character escape	Equivalent regular expression	Description
\S	[^\s]	Any character except a space, tab, newline, or return character.
\i	none	The set of characters allowed as the first character in an XML name.
\I	[^\i]	Not in the set of characters allowed as the first character in an XML name.
\c	none	The set of characters allowed in an XML name.
\C	[^\c]	Not in the set of characters allowed in an XML name.
\d	\p{Nd}	A decimal digit.
\D	[^\d]	Not a decimal digit.
\w	[#x0000-#x10FFFF]-[\p{P}\p{Z}\p{C}]	A word character, which includes the following *charProperty* categories: letters, marks, symbols, and numbers.
\W	[^\w]	A non-word character, which includes the following *charProperty* categories: punctuation, separators, and other.

\p{*charProperty*}
> Specifies a character in a category. The categories are listed in Table 40.

\P{*charProperty*}
> Specifies the complement of a character category. The categories are listed in Table 40.

Table 40. Supported values of charProperty

charProperty	Category	Description
L	Letters	All letters
Lu	Letters	Uppercase
Ll	Letters	Lowercase
Lt	Letters	Title case
Lm	Letters	Modifier
Lo	Letters	Other
M	Marks	All marks
Mn	Marks	Nonspacing
Mc	Marks	Spacing combining
Me	Marks	Enclosing
N	Numbers	All numbers
Nd	Numbers	Decimal digit
Nl	Numbers	Letter
No	Numbers	Other
P	Punctuation	All punctuation
Pc	Punctuation	Connector
Pd	Punctuation	Dash

Table 40. Supported values of charProperty (continued)

charProperty	Category	Description
Ps	Punctuation	Open
Pe	Punctuation	Close
Pi	Punctuation	Initial quotation mark (can behave like Ps or Pe depending on usage)
Pf	Punctuation	Final quotation mark (can behave like Ps or Pe depending on usage)
Po	Punctuation	Other
Z	Separators	All separators
Zs	Separators	Space
Zl	Separators	Line
Zp	Separators	Paragraph
S	Symbols	All symbols
Sm	Symbols	Math
Sc	Symbols	Currency
Sk	Symbols	Modifier
So	Symbols	Other
C	Other	All others
Cc	Other	Control
Cf	Other	Format
Co	Other	Private use
Cn	Other	Not assigned

Note: Regular expressions are matched using a binary comparison. The default collation is not used.

Chapter 7. 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 41. Limits for XQuery numeric data types

Data type	Smallest value	Largest value	Additional limits
xs:float	-3.4028234663852886e+38	+3.4028234663852886e+38	Smallest positive value: +1.1754943508222875e-38 Largest negative value: -1.1754943508222875e-38
xs:double	-1.7976931348623158e+308	+1.7976931348623158e+308	Smallest positive value: +2.2250738585072014e-308 Largest negative value: +2.2250738585072014e-308
xs:decimal	Not available	Not available	Largest decimal precision: 31 digits
xs:integer	-9 223 372 036 854 775 808	+9 223 372 036 854 775 807	
xs:nonPositiveInteger	-9 223 372 036 854 775 808	0	
xs:negativeInteger	-9 223 372 036 854 775 808	-1	
xs:long	-9 223 372 036 854 775 808	9 223 372 036 854 775 807	
xs:int	-2 147 483 648	+2 147 483 647	
xs:short	-32 768	+32 767	
xs:byte	-128	+127	
xs:nonNegativeInteger	0	+9 223 372 036 854 775 807	
xs:unsignedLong	0	+9 223 372 036 854 775 807	
xs:unsignedInt	0	4 294 967 295	
xs:unsignedShort	0	+65 535	
xs:unsignedByte	0	+255	
xs:positiveInteger	+1	+9 223 372 036 854 775 807	

Table 42. Limits for XQuery date, time, and duration data types

Data type	Smallest value	Largest value
xs:duration	-P83333333333333Y3M11574074074DT1H46M39.999999S	P83333333333333Y3M11574074074DT1H46M39.999999S
xdt:yearMonthDuration	-P83333333333333Y3M	P83333333333333Y3M
xdt:dayTimeDuration	-P11574074074DT1H46M39.999999S	P11574074074DT1H46M39.999999S
xs:dateTime	0001-01-01T00:00:00.000000Z	9999-12-31T23:59:59.999999Z
xs:date	0001-01-01Z	9999-12-31Z
xs:time	00:00:00Z	23:59:59Z
xs:gDay	01Z	31Z
xs:gMonth	01Z	12Z

Table 42. Limits for XQuery date, time, and duration data types (continued)

Data type	Smallest value	Largest value
xs:gYear	0001Z	9999Z
xs:gYearMonth	0001-01Z	9999-12Z
xs:gMonthDay	01-01Z	12-31Z

Note: DB2 XQuery provides no support for negative dates or negative times.

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.

Appendix A. Overview of the DB2 technical information

DB2 technical information is available in multiple formats that can be accessed in multiple ways.

DB2 technical information is available through the following tools and methods:
- DB2 Information Center
  - Topics (Task, concept and reference topics)
  - Sample programs
  - Tutorials
- DB2 books
  - PDF files (downloadable)
  - PDF files (from the DB2 PDF DVD)
  - printed books
- Command-line help
  - Command help
  - Message help

Note: The DB2 Information Center topics are updated more frequently than either the PDF or the hardcopy books. To get the most current information, install the documentation updates as they become available, or refer to the DB2 Information Center at ibm.com.

You can access additional DB2 technical information such as technotes, white papers, and IBM Redbooks® publications online at ibm.com. Access the DB2 Information Management software library site at http://www.ibm.com/software/data/sw-library/.

Documentation feedback

We value your feedback on the DB2 documentation. If you have suggestions for how to improve the DB2 documentation, send an email to db2docs@ca.ibm.com. The DB2 documentation team reads all of your feedback, but cannot respond to you directly. Provide specific examples wherever possible so that we can better understand your concerns. If you are providing feedback on a specific topic or help file, include the topic title and URL.

Do not use this email address to contact DB2 Customer Support. If you have a DB2 technical issue that the documentation does not resolve, contact your local IBM service center for assistance.

DB2 technical library in hardcopy or PDF format

The following tables describe the DB2 library available from the IBM Publications Center at www.ibm.com/e-business/linkweb/publications/servlet/pbi.wss. English and translated DB2 Version 10.1 manuals in PDF format can be downloaded from www.ibm.com/support/docview.wss?rs=71&uid=swg2700947.

Although the tables identify books available in print, the books might not be available in your country or region.

The form number increases each time a manual is updated. Ensure that you are reading the most recent version of the manuals, as listed below.

Note: The *DB2 Information Center* is updated more frequently than either the PDF or the hard-copy books.

Table 43. DB2 technical information

Name	Form Number	Available in print	Last updated
Administrative API Reference	SC27-3864-00	Yes	April, 2012
Administrative Routines and Views	SC27-3865-00	No	April, 2012
Call Level Interface Guide and Reference Volume 1	SC27-3866-00	Yes	April, 2012
Call Level Interface Guide and Reference Volume 2	SC27-3867-00	Yes	April, 2012
Command Reference	SC27-3868-00	Yes	April, 2012
Database Administration Concepts and Configuration Reference	SC27-3871-00	Yes	April, 2012
Data Movement Utilities Guide and Reference	SC27-3869-00	Yes	April, 2012
Database Monitoring Guide and Reference	SC27-3887-00	Yes	April, 2012
Data Recovery and High Availability Guide and Reference	SC27-3870-00	Yes	April, 2012
Database Security Guide	SC27-3872-00	Yes	April, 2012
DB2 Workload Management Guide and Reference	SC27-3891-00	Yes	April, 2012
Developing ADO.NET and OLE DB Applications	SC27-3873-00	Yes	April, 2012
Developing Embedded SQL Applications	SC27-3874-00	Yes	April, 2012
Developing Java Applications	SC27-3875-00	Yes	April, 2012
Developing Perl, PHP, Python, and Ruby on Rails Applications	SC27-3876-00	No	April, 2012
Developing User-defined Routines (SQL and External)	SC27-3877-00	Yes	April, 2012
Getting Started with Database Application Development	GI13-2046-00	Yes	April, 2012

Table 43. DB2 technical information (continued)

Name	Form Number	Available in print	Last updated
Getting Started with DB2 Installation and Administration on Linux and Windows	GI13-2047-00	Yes	April, 2012
Globalization Guide	SC27-3878-00	Yes	April, 2012
Installing DB2 Servers	GC27-3884-00	Yes	April, 2012
Installing IBM Data Server Clients	GC27-3883-00	No	April, 2012
Message Reference Volume 1	SC27-3879-00	No	April, 2012
Message Reference Volume 2	SC27-3880-00	No	April, 2012
Net Search Extender Administration and User's Guide	SC27-3895-00	No	April, 2012
Partitioning and Clustering Guide	SC27-3882-00	Yes	April, 2012
pureXML Guide	SC27-3892-00	Yes	April, 2012
Spatial Extender User's Guide and Reference	SC27-3894-00	No	April, 2012
SQL Procedural Languages: Application Enablement and Support	SC27-3896-00	Yes	April, 2012
SQL Reference Volume 1	SC27-3885-00	Yes	April, 2012
SQL Reference Volume 2	SC27-3886-00	Yes	April, 2012
Text Search Guide	SC27-3888-00	Yes	April, 2012
Troubleshooting and Tuning Database Performance	SC27-3889-00	Yes	April, 2012
Upgrading to DB2 Version 10.1	SC27-3881-00	Yes	April, 2012
What's New for DB2 Version 10.1	SC27-3890-00	Yes	April, 2012
XQuery Reference	SC27-3893-00	No	April, 2012

Table 44. DB2 Connect-specific technical information

Name	Form Number	Available in print	Last updated
DB2 Connect Installing and Configuring DB2 Connect Personal Edition	SC27-3861-00	Yes	April, 2012
DB2 Connect Installing and Configuring DB2 Connect Servers	SC27-3862-00	Yes	April, 2012
DB2 Connect User's Guide	SC27-3863-00	Yes	April, 2012

Displaying SQL state help from the command line processor

DB2 products return an SQLSTATE value for conditions that can be the result of an SQL statement. SQLSTATE help explains the meanings of SQL states and SQL state class codes.

Procedure

To start SQL state help, open the command line processor and enter:

    ? *sqlstate* or ? *class code*

where *sqlstate* represents a valid five-digit SQL state and *class code* represents the first two digits of the SQL state.
For example, ? 08003 displays help for the 08003 SQL state, and ? 08 displays help for the 08 class code.

Accessing different versions of the DB2 Information Center

Documentation for other versions of DB2 products is found in separate information centers on ibm.com®.

About this task

For DB2 Version 10.1 topics, the *DB2 Information Center* URL is http://publib.boulder.ibm.com/infocenter/db2luw/v10r1.

For DB2 Version 9.8 topics, the *DB2 Information Center* URL is http://publib.boulder.ibm.com/infocenter/db2luw/v9r8/.

For DB2 Version 9.7 topics, the *DB2 Information Center* URL is http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/.

For DB2 Version 9.5 topics, the *DB2 Information Center* URL is http://publib.boulder.ibm.com/infocenter/db2luw/v9r5.

For DB2 Version 9.1 topics, the *DB2 Information Center* URL is http://publib.boulder.ibm.com/infocenter/db2luw/v9/.

For DB2 Version 8 topics, go to the *DB2 Information Center* URL at: http://publib.boulder.ibm.com/infocenter/db2luw/v8/.

Updating the DB2 Information Center installed on your computer or intranet server

A locally installed DB2 Information Center must be updated periodically.

Before you begin

A DB2 Version 10.1 Information Center must already be installed. For details, see the "Installing the DB2 Information Center using the DB2 Setup wizard" topic in *Installing DB2 Servers*. All prerequisites and restrictions that applied to installing the Information Center also apply to updating the Information Center.

About this task

An existing DB2 Information Center can be updated automatically or manually:

- Automatic updates update existing Information Center features and languages.
  One benefit of automatic updates is that the Information Center is unavailable
  for a shorter time compared to during a manual update. In addition, automatic
  updates can be set to run as part of other batch jobs that run periodically.

- Manual updates can be used to update existing Information Center features and
  languages. Automatic updates reduce the downtime during the update process,
  however you must use the manual process when you want to add features or
  languages. For example, a local Information Center was originally installed with
  both English and French languages, and now you want to also install the
  German language; a manual update will install German, as well as, update the
  existing Information Center features and languages. However, a manual update
  requires you to manually stop, update, and restart the Information Center. The
  Information Center is unavailable during the entire update process. In the
  automatic update process the Information Center incurs an outage to restart the
  Information Center after the update only.

This topic details the process for automatic updates. For manual update
instructions, see the "Manually updating the DB2 Information Center installed on
your computer or intranet server" topic.

Procedure

To automatically update the DB2 Information Center installed on your computer or
intranet server:

1. On Linux operating systems,
   a. Navigate to the path where the Information Center is installed. By default,
      the DB2 Information Center is installed in the `/opt/ibm/db2ic/V10.1`
      directory.
   b. Navigate from the installation directory to the `doc/bin` directory.
   c. Run the `update-ic` script:

      `update-ic`

2. On Windows operating systems,
   a. Open a command window.
   b. Navigate to the path where the Information Center is installed. By default,
      the DB2 Information Center is installed in the `<Program Files>\IBM\DB2
      Information Center\Version 10.1` directory, where `<Program Files>`
      represents the location of the Program Files directory.
   c. Navigate from the installation directory to the `doc\bin` directory.
   d. Run the `update-ic.bat` file:

      `update-ic.bat`

Results

The DB2 Information Center restarts automatically. If updates were available, the
Information Center displays the new and updated topics. If Information Center
updates were not available, a message is added to the log. The log file is located in
`doc\eclipse\configuration` directory. The log file name is a randomly generated
number. For example, `1239053440785.log`.

Manually updating the DB2 Information Center installed on your computer or intranet server

If you have installed the DB2 Information Center locally, you can obtain and install documentation updates from IBM.

About this task

Updating your locally installed *DB2 Information Center* manually requires that you:

1. Stop the *DB2 Information Center* on your computer, and restart the Information Center in stand-alone mode. Running the Information Center in stand-alone mode prevents other users on your network from accessing the Information Center, and allows you to apply updates. The Workstation version of the DB2 Information Center always runs in stand-alone mode. .

2. Use the Update feature to see what updates are available. If there are updates that you must install, you can use the Update feature to obtain and install them

   Note: If your environment requires installing the *DB2 Information Center* updates on a machine that is not connected to the internet, mirror the update site to a local file system by using a machine that is connected to the internet and has the *DB2 Information Center* installed. If many users on your network will be installing the documentation updates, you can reduce the time required for individuals to perform the updates by also mirroring the update site locally and creating a proxy for the update site.
   If update packages are available, use the Update feature to get the packages. However, the Update feature is only available in stand-alone mode.

3. Stop the stand-alone Information Center, and restart the *DB2 Information Center* on your computer.

Note: On Windows 2008, Windows Vista (and higher), the commands listed later in this section must be run as an administrator. To open a command prompt or graphical tool with full administrator privileges, right-click the shortcut and then select **Run as administrator**.

Procedure

To update the *DB2 Information Center* installed on your computer or intranet server:

1. Stop the *DB2 Information Center*.
   - On Windows, click **Start** > **Control Panel** > **Administrative Tools** > **Services**. Then right-click **DB2 Information Center** service and select **Stop**.
   - On Linux, enter the following command:
     ```
     /etc/init.d/db2icdv10 stop
     ```

2. Start the Information Center in stand-alone mode.
   - On Windows:
     a. Open a command window.
     b. Navigate to the path where the Information Center is installed. By default, the *DB2 Information Center* is installed in the `Program_Files`\IBM\DB2 Information Center\Version 10.1 directory, where *Program_Files* represents the location of the `Program Files` directory.
     c. Navigate from the installation directory to the `doc\bin` directory.
     d. Run the `help_start.bat` file:

```
help_start.bat
```

- On Linux:

  a. Navigate to the path where the Information Center is installed. By default, the *DB2 Information Center* is installed in the `/opt/ibm/db2ic/V10.1` directory.

  b. Navigate from the installation directory to the `doc/bin` directory.

  c. Run the `help_start` script:

  ```
  help_start
  ```

The systems default Web browser opens to display the stand-alone Information Center.

3. Click the **Update** button (![icon]). (JavaScript must be enabled in your browser.) On the right panel of the Information Center, click **Find Updates**. A list of updates for existing documentation displays.

4. To initiate the installation process, check that the selections you want to install, then click **Install Updates**.

5. After the installation process has completed, click **Finish**.

6. Stop the stand-alone Information Center:

   - On Windows, navigate to the `doc\bin` directory within the installation directory, and run the `help_end.bat` file:

   ```
   help_end.bat
   ```

   Note: The `help_end` batch file contains the commands required to safely stop the processes that were started with the `help_start` batch file. Do not use `Ctrl-C` or any other method to stop `help_start.bat`.

   - On Linux, navigate to the `doc/bin` directory within the installation directory, and run the `help_end` script:

   ```
   help_end
   ```

   Note: The `help_end` script contains the commands required to safely stop the processes that were started with the `help_start` script. Do not use any other method to stop the `help_start` script.

7. Restart the *DB2 Information Center*.

   - On Windows, click **Start** > **Control Panel** > **Administrative Tools** > **Services**. Then right-click **DB2 Information Center** service and select **Start**.

   - On Linux, enter the following command:

   ```
   /etc/init.d/db2icdv10 start
   ```

Results

The updated *DB2 Information Center* displays the new and updated topics.

DB2 tutorials

The DB2 tutorials help you learn about various aspects of DB2 database products. Lessons provide step-by-step instructions.

Before you begin

You can view the XHTML version of the tutorial from the Information Center at http://publib.boulder.ibm.com/infocenter/db2luw/v10r1/.

Some lessons use sample data or code. See the tutorial for a description of any prerequisites for its specific tasks.

DB2 tutorials

To view the tutorial, click the title.

"pureXML®" in *pureXML Guide*
> Set up a DB2 database to store XML data and to perform basic operations with the native XML data store.

DB2 troubleshooting information

A wide variety of troubleshooting and problem determination information is available to assist you in using DB2 database products.

DB2 documentation
> Troubleshooting information can be found in the *Troubleshooting and Tuning Database Performance* or the Database fundamentals section of the *DB2 Information Center*, which contains:
> * Information about how to isolate and identify problems with DB2 diagnostic tools and utilities.
> * Solutions to some of the most common problem.
> * Advice to help solve other problems you might encounter with your DB2 database products.

IBM Support Portal
> See the IBM Support Portal 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 IBM Support Portal at http://www.ibm.com/support/entry/ portal/Overview/Software/Information_Management/ DB2_for_Linux,_UNIX_and_Windows

Terms and conditions

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability: These terms and conditions are in addition to any terms of use for the IBM website.

Personal use: You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial use: You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Rights: Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

IBM® Trademarks: IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at www.ibm.com/legal/copytrade.shtml

Appendix B. Notices

This information was developed for products and services offered in the U.S.A. Information about non-IBM products is based on information available at the time of first publication of this document and is subject to change.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information about 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 grant 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 character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 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, changes, or both in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to websites not owned by IBM are provided for convenience only and do not in any manner serve as an endorsement of those

websites. The materials at those websites are not part of the materials for this IBM product and use of those websites 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
    U59/3600
    3600 Steeles Avenue East
    Markham, Ontario   L3R 9Z7
    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 contains 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. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample 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

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

The following terms are trademarks or registered trademarks of other companies
- Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.
- Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle, its affiliates, or both.
- UNIX is a registered trademark of The Open Group in the United States and other countries.
- Intel, Intel logo, Intel Inside, Intel Inside logo, Celeron, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.
- Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Index

A

abbreviated syntax 66
abs function 139
adding nodes 118
adjust-date-to-timezone function 133
adjust-dateTime-to-timezone function 135
adjust-time-to-timezone function 137
and operator 79
anyAtomicType data type 27
anySimpleType data type 27
anyType data type 27
anyURI data type 27
arithmetic expressions 72
atomic types 17
atomic values 5
atomization 55
attribute axis 63
attribute nodes 9
attributes
   computed constructors
      description 90
   constructing 90
   namespace declaration 85
avg function 139
axes
   abbreviated syntax 66
   in path expressions 63
axis steps
   in path expressions 63
   node tests 64

B

base64Binary data type 27
binary data types 19
boolean data type 19, 28
Boolean function 140
Boolean functions
   XQuery 127
boundary whitespace
   declaration 44
   in direct element constructors 86
boundary-space declarations 44
built-in data types, constructors for 23
built-in functions 127
byte data type 28

C

case sensitivity, query language 14
cast expressions 108
castable expressions 109
casting
   XQuery
      castable expressions 109
casting, data types 24
ceiling function 141
changing node names 121
character references 59
child axis 63

codepoints-to-string function 142
combining updating expressions 111
comment nodes 10
comments
   computed constructors 94
   constructing, overview 94
   direct constructors 94
   query language, description 14
compare function 143
comparison expressions
   general 76
   nodes 78
   overview 74
   value 74
computed attribute constructors 90
computed constructors
   attribute
      description 90
   comment 94
   description 80
   element 89
   processing instruction 93
concat function 144
conditional expressions 106
construction declarations 45
constructors
   attribute
      description 90
   built-in types 23
   computed attribute 90
   computed comment 94
   computed element 89
   computed processing instruction 93
   direct comment 94
   direct element
      description 82
   direct processing instruction 93
   document node 91
   enclosed expressions 81
   in-scope namespaces 88
   namespace declaration attributes 85
   processing instruction 92
   text node 92
   XML 80
contains function 144
context item expressions 60
context of expressions 51
copy-namespace declarations 45
count function 145
current-date function 145
current-dateTime function 146
current-local-date function
   description 146
current-local-dateTime function
   description 146
current-local-time function
   description 147
current-time function 147

D

IBM®

Printed in USA

Spine information:

IBM DB2 10.1 for Linux, UNIX, and Windows

XQuery Reference

IBM