

XQuery Scripts

An XQuery script consists of:

1. A Version Declaration

xquery version StringLiteral

followed, optionally, by:

encoding StringLiteral

followed, optionally, by a semicolon (";").

2. If an XQuery script is a Library Module, then it's module namespace declaration comes next:

module namespace NCName = URILiteral ;

3. Default Declarations and Imports:

zero or more of:

declare default element namespace URILiteral ;

declare default function namespace URILiteral ;

declare boundary-space preserve ;

declare boundary-space strip ;

declare default collation URILiteral ;

declare base-uri URILiteral ;

declare construction strip ;

declare construction preserve ;

declare ordering ordered ;

declare ordering unordered ;

declare default order empty greatest ;

declare default order empty least ;

declare copy-namespaces preserve , inherit ;

declare copy-namespaces preserve , no-inherit ;

declare copy-namespaces no-preserve , inherit ;

declare copy-namespaces no-preserve , no-inherit ;

declare namespace NCName = URILiteral ;

import schema namespace NCName = URILiteralList ;

import schema default element namespace URILiteralList ;

import schema URILiteralList ;

import module namespace NCName = URILiteralList ;

import module URILiteralList ;

XQuery 1.0:

<http://www.w3.org/TR/xquery/>

4. Variable, Function and Option Declarations:

zero or more of:

declare variable VariableDeclaration := ExprSingle ;

declare variable VariableDeclaration **external** ;

declare function QName
ParameterDeclarations ;

declare function QName
ParameterDeclarations
external ;

declare function QName
ParameterDeclarations **as**
SequenceType **external** ;

declare option QName StringLiteral ;

where ParameterDeclarations is one of:

() (i.e. empty if no parameters)

(VariableDeclaration) (for one parameter)

(VariableDeclaration , ...) (when two or more)

where VariableDeclaration is one of:

\$QName

\$QName **as** SequenceType

and where URILiteralList is one of:

URILiteral

URILiteral **at** URILiteral

URILiteral **at** URILiteral , ... (two or more)

5. Finally, if the XQuery script is a Main module, not a Library module, an XQuery expression is required to specify the query being made:

Expr

Creating Sequences

Create a sequence from a list of items:

Expr , ...

Note: A sequence list must usually be parenthesized.

Repeat over one or more sequences, returning a sequence of results:

for VariableBinding , ... **return** Expr

Create a numeric sequences, from lower bound to upper bound:

Expr **to** Expr

All the items appearing in either sequence:

Expr **union** Expr Expr | Expr

Only items appearing in both sequences:

Expr **intersect** Expr

All items in the first sequence not in second:

Expr **except** Expr

Arithmetic Expressions

+ Expr Expr + Expr

- Expr Expr - Expr

Expr * Expr Expr **div** Expr

Expr **idiv** Expr Expr **mod** Expr

Type Modification Expressions

Use as without converting:

Expr **treat as** SequenceType

Use as, converting as needed and doable:

Expr **cast as** AtomicType

Expr **cast as** AtomicType?

Simple Expressions

\$ VarName . (one dot: self)

() (Expr)

QName (Expr , ...) QName ()

IntegerLiteral DecimalLiteral

DoubleLiteral StringLiteral

Validating Nodes

validate { Expr } (defaults to **strict**)

validate lax { Expr }

validate strict { Expr }

Ordering Mode for Sequences

ordered { Expr }

unordered { Expr }

Implementation-Defined Instructions

(# QName ... #) ... { OptionalExpr }

Path Expressions

/ Top level, document root

/ Step At top level

Step Relative to current node

// Step Anywhere within document

Path / Step Immediately within Path

Path // Step Anywhere within Path

Where a Step is one of:

Expr

AxisName :: NameTest

AxisName :: KindTest

@NameTest (attribute test)

NameTest (child element test)

KindTest (child node test)

.. (two dots: parent test)

Followed by zero or more predicates:

[Expr]

Where an AxisName is one of:

ancestor **ancestor-or-self**

attribute **child**

descendant **descendant-or-self**

following **following-sibling**

namespace **parent**

preceding **preceding-sibling**

self

A NameTest is one of:

QName

NCName:* *:NCName

And a KindTest is one of:

attribute (AttributeName)

attribute (AttributeName , TypeName)

attribute (* , TypeName)

attribute (*)

attribute ()

comment ()

document-node (element ...)

document-node (schema-element ...)

document-node ()

element (ElementName)

element (ElementName , TypeName)

element (* , TypeName)

element (*)

element ()

node ()

processing-instruction (NCName)

processing-instruction (StringLiteral)

processing-instruction ()

schema-attribute (AttributeName)

schema-element (ElementName)

text ()

Testing

Select based on the type of an expression (one or more **cases** plus a **default**):

typeswitch (Expr) **case** ... **default** ...

where **case** and **default** are:

case SequenceType **return** Expr

case \$VarName **as** SequenceType **return** Expr

default return Expr

default \$VarName **return** Expr

Test if the condition is satisfied for at least one combination of the bound expressions:

some VariableBinding , ... **satisfies** Expr

Test if the condition is satisfied for all of the bound expressions:

every VariableBinding , ... **satisfies** Expr

where a VariableBinding is:

\$VarName **in** Expr

\$VarName **as** SequenceType **in** Expr

Select one or the other of two possibilities:

if (Expr) **then** Expr **else** Expr

Either or both of two tests:

Expr **or** Expr Expr **and** Expr

Test if they are the same node:

Expr **is** Expr

Test if a node appears before or after another:

Expr << Expr Expr >> Expr

Test an expression's dynamic type:

Expr **instance of** SequenceType

Test if an expression can be converted to a type:

Expr **castable as** AtomicType

Expr **castable as** AtomicType?

Compare two item values:

Expr **eq** Expr Expr **ne** Expr

Expr **lt** Expr Expr **le** Expr

Expr **gt** Expr Expr **ge** Expr

Compare all items in one sequence to all items in a second, and return if true for any pair of values:

Expr = Expr Expr != Expr

Expr < Expr Expr <= Expr

Expr > Expr Expr >= Expr

Names and Types

VarName AttributeName ElementName
TypeName AtomicType

are all XML QNames, with or without a colon-separated prefix.

A SequenceType is one of:

empty-sequence () KindTest
item () AtomicType

Where KindTest, **item()** or AtomicType can be optionally followed by:

? (may be empty sequence)
+ (is a non-empty sequence of the type)
* (is a sequence of the type, empty or not)

Operator Precedence:

1 , (comma)
2 **for let some every if typeswitch**
3 **or**
4 **and**
5 = != < <= > >=
6 **eq ne lt le gt ge is << >>**
7 (two-argument) + -
8 * **div idiv mod**
9 **union |**
10 **intersect except**
11 **instance of**
12 **treat as**
13 **castable as**
14 **cast as**
15 (one-argument) + -
16 / //
17 step node-test \$ name
 (Expr) function-call literal
validate (# ... #) constructor
ordered **unordered**

Predefined Namespace Names:

xml = <http://www.w3.org/XML/1998/namespace>
xs = <http://www.w3.org/2001/XMLSchema>
xsi =
<http://www.w3.org/2001/XMLSchema-instance>
fn = <http://www.w3.org/2005/xpath-functions>
local =
<http://www.w3.org/2005/xquery-local-functions>
2008-07-21

XQuery 1.0 Quick Reference

See also the "XQuery 1.0 & XPath 2.0 Functions & Operators Quick Reference"

Sam Wilmott
sam@wilmott.ca
<http://www.wilmott.ca>

and

Mulberry Technologies, Inc.
17 West Jefferson Street, Suite 207
Rockville, MD 20850 USA
Phone: +1 301/315-9631
Fax: +1 301/315-8285
info@mulberrytech.com
<http://www.mulberrytech.com>



© 2007-2008 Sam Wilmott and
Mulberry Technologies, Inc.

FLWOR Expressions

FLWOR Expressions start with one or more **for** or **let**:

for SequenceVariableBinding , ...

let AssignedVariableBinding , ...

followed by:

where Expr (optional)
OrderingInfo , ... (one or more, optional)
return Expr

where SequenceVariableBinding is one of:

\$VarName **in** Expr

\$VarName **as** SequenceType **in** Expr

\$VarName **at** \$ VarName **in** Expr

\$VarName **as** SequenceType **at** \$ VarName **in** Expr

where AssignedVariableBinding is one of:

\$VarName := Expr

\$VarName **as** SequenceType := Exp

and where OrderingInfo consists of, in order:

stable (optional)

order Expr (optional)

ascending or **descending** (optional)

empty greatest or **empty least** (optional)

collation URILiteral (optional)

Constructors

< QName ... />

< QName ... > ... </ QName >

<![CDATA[...]]>

<!-- ... -->

<? PITarget ... ?>

document { Expr }

element QName { OptionalExpr }

element { Expr } { OptionalExpr }

attribute QName { OptionalExpr }

attribute { Expr } { OptionalExpr }

text { Expr }

comment { Expr }

processing-instruction NCName { OptionalExpr }

processing-instruction { Expr } { OptionalExpr }

Within a constructor's attribute values and element content, literal "{" and "}" need doubling. Anything within single "{" ... "}" is evaluated as an Expr.