MarkLogic Server

XQuery Reference Guide

Release 4.0
September, 2008

Last Revised: 4.0-4, March, 2009

Copyright

Table of Contents

XQuery Reference Guide

Copyright 2

1.0 About This XQuery Guide 6

2.0 XQuery Dialects in MarkLogic Server 7
2.1 Overview of the XQuery Dialects 7
2.1.1 MarkLogic Server Enhanced (XQuery 1.0-ml) 7
2.1.2 MarkLogic Server 3.2 Compatibility (XQuery 0.9-ml) 8
2.1.3 Strict (XQuery 1.0) 8
2.2 Rules For Combining the Dialects 8
2.3 Strategies For Migrating Code to Enhanced Dialect 9
2.3.1 When To Migrate XQuery Code 9
2.3.2 XQuery Changes From Previous MarkLogic Server Releases 10
2.3.3 Inheriting the Default XQuery Version From the App Server 11
2.4 Specifying the XQuery Dialect in the Prolog 11
2.4.1 Porting 3.2 (0.9-ml) XQuery Code to Enhanced (1.0-ml) 11

3.0 MarkLogic Server Enhanced XQuery Language 14
3.1 try/catch Expression 14
3.2 Function Mapping 15
3.2.1 Understanding Function Mapping 15
3.2.2 Enabling or Disabling Function Mapping 16
3.3 Semi-Colon as Transaction Separator 16
3.4 Private Function and Variable Definitions 17
3.5 Functions With Side Effects 17
3.6 Shorthand Positional Predicate Syntax 17
3.7 Binary Node Constructor and Node Test 18
3.8 Implementation-Defined Semantics 18
3.8.1 Automatic Namespace Imports for Predefined Namespaces 18
3.8.2 External Variables 19
3.8.3 Collations 19
3.8.4 Implementation-Defined Primitive XQuery Types 20
3.8.5 Decimal Precision at Least 18 Digits, and is Not Configurable 20
3.8.6 Library Modules Default Function Namespace Defaults to Library Namespace 20

4.0 XQuery Language 21
4.1 Expressions Return Items 21
4.2 XML and XQuery 22
4.2.1 Direct Element Constructors: Switching Between XQuery and XML Using Curly Braces 22
4.2.2 Computed Element and Attribute Constructors 23
4.2.3 Returning XML From an XQuery Program 23
4.3 XQuery Modules 24
4.3.1 XQuery Version Declaration 24
4.3.2 Main Modules 25
4.3.3 Library Modules 25
4.4 XQuery Prolog 26
4.4.1 Importing Modules or Schemas 26
4.4.2 Declaring Namespaces 26
4.4.3 Declaring Functions 27
4.4.4 Declaring Variables 27
4.4.5 Declaring a Default Collation 27
4.5 XQuery Comments 28
4.6 XQuery Expressions 28
4.6.1 XPath Expressions 28
4.6.2 FLWOR Expressions 28
4.6.2.1 The for Clause 29
4.6.2.2 The let Clause 30
4.6.2.3 The where Clause 31
4.6.2.4 The order by Clause 32
4.6.2.5 The return Clause 32
4.6.3 The typeswitch Expression 33
4.6.4 The if Expression 34
4.6.5 Quantified Expressions (some/every ... satisfies ...) 35
4.6.6 Validate Expression 36
4.7 XQuery Comparison Operators 36
4.7.1 Node Comparison Operators 37
4.7.2 Sequence and Item Operators 38
4.7.2.1 Sequence Operators 38
4.7.2.2 Item Operators 39

5.0 XPath Quick Reference 40
5.1 Path Expressions 40
5.2 XPath Axes and Syntax 41
5.3 XPath 2.0 Functions 42

6.0 Understanding XML Namespaces in XQuery 43
6.1 XML QNames, Local Names, and Namespaces 43
6.2 Everything Is In a Namespace 43
6.3 XML Data Model Versus Serialized XML 44

6.3.1 XQuery Accesses the XML Data Model 44
6.3.2 Serialized XML: Human-Readable With Angle Brackets 44
6.3.3 Understanding Namespace Inheritance With the xmlns Attribute 46
6.4 Declaring a Default Element Namespace in XQuery 47
6.5 Tips For Constructing QNames 47
6.6 Predefined Namespace Prefixes for Each Dialect 48
6.6.1 1.0-ml Predefined Namespaces 48
6.6.2 1.0 Predefined Namespaces 50
6.6.3 0.9-ml Predefined Namespaces 51

7.0 Application Programming in XQuery 53
7.1 Design Patterns 53
7.2 Using Functions 53
7.2.1 Creating Reusable and Modular Code 54
7.2.2 Recursive Functions 54
7.3 Search Functions 54
7.4 Updates and Transactions 55
7.5 HTTP App Server Functions 55
7.6 Additional Resources 55
7.6.1 MarkLogic Server Documentation 55
7.6.2 XQuery Use Cases 56
7.6.3 Other Publications 56

8.0 Technical Support 57

1.0 About This XQuery Guide

This *XQuery Reference Guide* briefly describes some of the basics of the XQuery language, but describes more thoroughly the MarkLogic Server implementation of XQuery, including many of the important extensions to the language implemented in MarkLogic Server.

The next two chapters ("XQuery Dialects in MarkLogic Server" on page 7 and "MarkLogic Server Enhanced XQuery Language" on page 14) focus on the MarkLogic Server-specific aspects of the XQuery language. If you prefer to start with the more generic aspects of the XQuery language before moving to the MarkLogic Server-specific parts, start with "XQuery Language" on page 21.

Specifically, this guide covers:

- The different dialects of XQuery supported in MarkLogic Server (see "XQuery Dialects in MarkLogic Server" on page 7).
- Mark Logic extensions to the XQuery language (see "MarkLogic Server Enhanced XQuery Language" on page 14).
- An overview of the basic syntax of the XQuery language (see "XQuery Language" on page 21).
- A brief description of XPath syntax (see "XPath Quick Reference" on page 40).
- An introduction to how namespaces work in XML and XQuery (see "Understanding XML Namespaces in XQuery" on page 43).
- Some information on how XQuery and MarkLogic Server is used as an application development programming language (see "Application Programming in XQuery" on page 53).

2.0 XQuery Dialects in MarkLogic Server

The XQuery specification is a formal recommendation from the W3C XQuery Working Group. MarkLogic Server 4.0 implements the W3C XQuery 1.0 Recommendation (http://www.w3.org/TR/xquery/). To maximize compatibility with MarkLogic Server 3.2 and to offer strict XQuery compliance to those who desire it, as well as to include extensions to the language to make it easier to build applications, MarkLogic Server supports three dialects of XQuery. This chapter describes these dialects, and includes the following sections:

- Overview of the XQuery Dialects
- Rules For Combining the Dialects
- Specifying the XQuery Dialect in the Prolog
- Strategies For Migrating Code to Enhanced Dialect

2.1 Overview of the XQuery Dialects

MarkLogic Server supports three dialects separate dialects of XQuery:

- MarkLogic Server Enhanced (XQuery 1.0-ml)
- MarkLogic Server 3.2 Compatibility (XQuery 0.9-ml)
- Strict (XQuery 1.0)

You can use library modules from different dialects together, as described in "Rules For Combining the Dialects" on page 8. Each dialect has a different set of pre-defined namespaces, as described in "Predefined Namespace Prefixes for Each Dialect" on page 48.

2.1.1 MarkLogic Server Enhanced (XQuery 1.0-ml)

For a module to use the MarkLogic Server enhanced dialect, use the following for the XQuery version declaration on the first line of the XQuery module:

```
xquery version "1.0-ml";
```

Note the semi-colon at the end of the declaration, which is required in `1.0-ml`. The enhanced dialect has the XQuery 1.0 syntax and also includes various extensions to the language such as `try/catch`. This dialect is the default for new App Servers, and should be considered the preferred dialect for new applications. For more details on the enhanced `1.0-ml` dialect, see "MarkLogic Server Enhanced XQuery Language" on page 14.

2.1.2 MarkLogic Server 3.2 Compatibility (XQuery 0.9-ml)

For a module to use the MarkLogic Server 3.2 compatibility dialect, use the following for the XQuery version declaration on the first line of the XQuery module:

```
xquery version "0.9-ml"
```

Note there is no semi-colon at the end of the declaration for `0.9-ml`. The 3.2 compatibility dialect allows you to write code that you can use with both 3.2 and 4.0. Any code you have from 3.2 releases is equivalent to `0.9-ml`. To use that code in 4.0, the best practice is to add the `0.9-ml` XQuery declaration (shown above) as the first line of each XQuery module.

2.1.3 Strict (XQuery 1.0)

For a module to use the MarkLogic Server strict dialect, use the following for the XQuery version declaration on the first line of the XQuery module:

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

Note the semi-colon at the end of the declaration, which is required in `1.0`. The strict mode is for compatibility with other XQuery 1.0 processors; if you write a library in `1.0`, you can use it with MarkLogic Server and you can also use it with other conforming processors. Similarly, you can use modules that are written in standard XQuery with MarkLogic Server.

To use the MarkLogic Server built-in functions in `1.0`, you must import the namespace for the MarkLogic Server functions; there is no need to import a library for these built-in functions, but you do need to bind the namespace to a prefix. To use the `xdmp` functions in `1.0`, add prolog entries for the namespace bindings you are using in your query, as in the following example:

```
xquery version "1.0";
declare namespace xdmp = "http://marklogic.com/xdmp";

xdmp:version()
```

2.2 Rules For Combining the Dialects

MarkLogic Server has a very flexible way of combining the three XQuery dialects. You can import a library module written in any of the three dialects into any main or library module. For example, you might find an open source standards-compliant module that you found on the internet which is written in the strict XQuery 1.0 dialect. You can then import this module into any MarkLogic Server XQuery program, regardless of dialect, and then use those functions in your code.

When writing modules of different dialects, the best practice is to always use the XQuery version declaration as the first line, indicating which dialect the module is written in. That way, if the module is written in a different dialect than the default dialect for the App Server or the program, it will still work correctly (for details, see “Inheriting the Default XQuery Version From the App Server” on page 11).

2.3 Strategies For Migrating Code to Enhanced Dialect

If you are writing new XQuery code, the best practice is to use the `1.0-ml` dialect. If you are updating code that was written in previous versions of MarkLogic Server, you can consider if you want to migrate that code to `1.0-ml`. This section describes things to think about when migrating your application code and includes the following parts:

- When To Migrate XQuery Code
- XQuery Changes From Previous MarkLogic Server Releases
- Inheriting the Default XQuery Version From the App Server
- Porting 3.2 (0.9-ml) XQuery Code to Enhanced (1.0-ml)

2.3.1 When To Migrate XQuery Code

Because of the flexibility of how you can interoperably use the various XQuery dialects, it is really up to you when and how you migrate your XQuery code. The differences between the dialects are mostly syntax changes in the prolog, but there are also some other differences that might cause subtle changes in behavior. For details on the differences between the XQuery dialects in 3.2 (`0.9-ml`) and 4.0 (`1.0-ml`), see “XQuery Changes From Previous MarkLogic Server Releases” on page 10. When you decide to migrate XQuery code to `1.0-ml` (or to `1.0`), there are several ways you can go about it:

- Migrate an entire application all at once. This method gets everything over with at once, and therefore focuses the effort. If you have a relatively small amount of code to migrate, it might make sense to just go ahead and migrate it all at once.
- Migrate one module at a time. This method allows you to spread the migration work over a number of small tasks instead of one large task, and further allows you to test each module independently after migration. This technique is very flexible, as you can do a little bit at a time. A good first step for this one-by-one approach is to start by adding an XQuery `0.9-ml` declaration to the first line of each XQuery file. Then, as you migrate a module, you can change the declaration to `1.0-ml` and make any needed syntax changes to that module.

2.3.2 XQuery Changes From Previous MarkLogic Server Releases

While MarkLogic Server 4.0 includes a compatibility dialect to run your 3.2 code without changes (`0.9-ml`), the new enhanced mode offers several important improvements, so it is a good idea to migrate your code to the enhanced dialect (`1.0-ml`). Because you can mix modules in the old dialect with modules in the new, you can perform your migration one module at a time. This section highlights the major syntax and semantic changes between the XQuery used in MarkLogic Server 3.2 (`0.9-ml`) and MarkLogic Server 4.0 enhanced XQuery dialect (`1.0-ml`). Additionally, see the “Known Incompatibilities” section of the *Release Notes*. The changes include:

- Semi-colons (;) are now required at the end of each prolog declaration.
- Prolog declarations that previously used `define` now use `declare`.
- Variable declaration syntax is slightly different, and now uses the `:=` syntax (for details and an example, see “Declaring Variables” on page 27).
- Library module declarations now require the `namespace` keyword and a prefix for the namespace, for example:

  ```
  module namespace my = "my-namespace";
  ```

- Function declarations that return the empty sequence now require the empty sequence to be specified as follows:

  ```
  empty-sequence()
  ```

  In `0.9-ml`, you specify `empty()` for the empty sequence.

- Some of the effective boolean value rules have changed. Notably, the following returns true in `0.9-ml` and returns false in `1.0-ml` (and throws an exception in `1.0`):

  ```
  (: returns true in 0.9-ml, false in 1.0-ml, and
     throws XDMP-EFFBOOLVALUE in 1.0 :)
  fn:boolean((fn:false(), fn:false()))
  ```

  This change might affect applications that have `if/then/else` statements where the `if` test returns a sequence of boolean values. In these cases, you might see the `if` statement evaluating to `false` in cases where it previously evaluated to `true`, causing the `else` statement to be evaluated intead of the `then` statement.

- The namespace used for durations now uses the `xs` namespace prefix; previously it was the `xdt` prefix. Any code you have that uses the `xdt` namespace prefix will require a change to the `xs` prefix. For example, if you have code that uses `xdt:dayTimeDuration`, you should change that to `xs:dayTimeDuration`.

- `element()` tests in `0.9-ml` are equivalent to `schema-element()` test in `1.0` and `1.0-ml`. Any code you have with `element()` tests might not match some elements that previously matched. For example, substitution elements previously would match the base element name, but will now only match with `schema-element()` test in `1.0` and `1.0-ml`. For more information, see element() Test in 0.9-ml Equivalent to schema-element() Test in 1.0-ml in the *Release Notes*.
- Some changes to the XQuery standard functions. For example, there are subtle changes to `fn:avg` and `fn:sum`, `fn:error` has a different signature, and `fn:node-kind` does not exist in 1.0 and 1.0-ml (it is replaced by `xdmp:node-kind`).

2.3.3 Inheriting the Default XQuery Version From the App Server

Each App Server has a setting for the default XQuery version. Any requests against that App Server that do not have explicitly specify an XQuery version declaration are treated as the default XQuery version value. Because of the way a request inherits it default XQuery version from the App Server environment, requests without an explicit declaration can be treated differently by different App Servers (if the App Servers have different default XQuery values). Therefore, it is best practice to specify the XQuery version in each module.

The task server does not allow you to specify a default XQuery version, and if there is no explicit version declaration in the XQuery code evaluated on the task server, it defaults to `1.0-ml`. This makes it especially important to use XQuery version declarations in modules used by CPF or modules called from triggers. For details on CPF, see *Content Processing Framework*.

To ensure your code is always evaluated in the dialect in which you have written it, regardless of the context in which it is run, the best practice is to begin each XQuery module with a XQuery version declaration. For the syntax of the version declaration, see “XQuery Version Declaration” on page 24.

2.4 Specifying the XQuery Dialect in the Prolog

You specify the dialect for an XQuery module with a version declaration. The version declaration is optional, and comes before the prolog in an XQuery module. It is best practice to put the XQuery version declaration in your code as the first line in the module, as having it there ensures it will work as expected in any environment. For example, to specify `1.0-ml` as the XQuery version, begin your XQuery module with the following:

```
xquery version "1.0-ml";
```

2.4.1 Porting 3.2 (0.9-ml) XQuery Code to Enhanced (1.0-ml)

In most cases, porting any XQuery code used in 3.2 to the `1.0-ml` dialect will be easy and straightforward. The bulk of the differences are syntax changes in the prolog. As stated earlier, you do not need to port all of your code at one time. A sensible approach is to migrate your code one XQuery module at a time. This section outlines the basic steps you should follow when migrating your XQuery code.

The following are some basic steps to take when migrating 3.2 XQuery code (`0.9-ml`) to 4.0 (`1.0-ml`):

1. Add XQuery version declarations to all of your existing modules. For code written in 3.2, the declarations will be as follows:

   ```
   xquery version "0.9-ml"
   ```

2. Review the *Release Notes* for any incompatibilities.

3. For each module you migrate, change the version number string in the XQuery version declaration to `1.0-ml` and add a semi-colon to the line so it looks as follows

   ```
   xquery version "1.0-ml";
   ```

4. Change all of the prolog declarations to the 1.0 syntax (change `define` to `declare`, add semi-colons, and so on, as described in "XQuery Changes From Previous MarkLogic Server Releases" on page 10). For the prolog syntax, see "XQuery Prolog" on page 26, the W3C specification (http://www.w3.org/TR/xquery/#id-grammar), or a third-party XQuery book.

5. If you are modifying a main module and it has function declarations that are used in the same module, they must be declared in a namespace. The preferred way to put functions local to a main module is to prefix those functions definitions and function calls with the `local:` prefix, which is predefined.

6. If you have any durations that use the `xdt` namespace prefix, change the prefix to `xs` (for example, change `xdt:dayTimeDuration` to `xs:dayTimeDuration`).

7. If you are modifying a library module, all XQuery standard functions need to be prefixed with the `fn` namespace prefix. Alternately, you can declare the XQuery functions namespace as the default function namespace in the prolog as follows:

   ```
   declare default function namespace
               "http://www.w3.org/2005/xpath-functions";
   ```

   If you do declare the default function namespace, then you will also need to prefix your own function definitions with the prefix defined in your module definition. Note that you can no longer use the XPath functions namespace as the library module namespace.

8. If you are modifying a library module that is defined with the `fn` namespace URI, you must change the namespace URI of that module; you cannot use the URI bound to the `fn` namespace prefix as the URI for a library module in `1.0` or `1.0-ml`. If you do change the namespace URI of a library module, you must also change the URI in any `import module` statements in other modules that call the library.

9. Test the module and correct any syntax errors that occur.

10. After getting the module to run, test your code to make sure it behaves as it did before. Pay particular attention to parts of your code that might rely on boolean values that take boolean values of sequences, as those behave differently in `0.9-ml` and `1.0-ml` (see "XQuery Changes From Previous MarkLogic Server Releases" on page 10). Check for any changes due to function mapping, which is described in "Function Mapping" on page 15.

11. Repeat this process for other modules you want to migrate.

3.0 MarkLogic Server Enhanced XQuery Language

The default XQuery dialect in MarkLogic Server is *enhanced*. (`1.0-ml`) The enhanced dialect includes all of the features in the strict XQuery 1.0 dialect, and adds several other features to make it easier to use XQuery as a programming language with which to create applications. This chapter describes the features of the enhanced dialect and includes the following sections:

- try/catch Expression
- Function Mapping
- Semi-Colon as Transaction Separator
- Private Function and Variable Definitions
- Functions With Side Effects
- Shorthand Positional Predicate Syntax
- Binary Node Constructor and Node Test
- Implementation-Defined Semantics

For details on the XQuery language, see "XQuery Language" on page 21 and the W3C XQuery specification (http://www.w3.org/TR/xquery/).

3.1 try/catch Expression

The try/catch extension allows you to catch and handle exceptions. MarkLogic Server exceptions are thrown in XML format, and you can apply an XPath statement to the exception if there is a particular part you want to extract. The exception is bound to the variable in the `catch` clause.

try { *expression* **}** **catch (** *variable* **)** **{** *expression* **}**

The following code sample uses a try/catch block to catch exceptions upon loading a document, and prints out the filename if an exception occurs.

```
try {
     let $filename := "/space/myfile.xml"
     let $options := <options xmlns="xdmp:document-load">
                         <uri>/myfile.xml</uri>
                         <repair>none</repair>
                      </options>
     return xdmp:document-load($filename, $options)
     }
catch ($exception) {
"Problem loading file, received the following exception: ",
$exception }
```

3.2 Function Mapping

Function mapping is an extension to XQuery that allows you to pass a sequence to a function parameter that is typed to take a singleton item, and it will invoke that function once for each item in the sequence. This section describes function mapping and includes the following parts:

- Understanding Function Mapping
- Enabling or Disabling Function Mapping

3.2.1 Understanding Function Mapping

Function mapping is equivalent to iterating over the sequence like it was in a `for` clause of a FLWOR expression. The following is an example of function mapping:

```
xquery version "1.0-ml";

declare function local:print-word ($word as xs:string) { $word };

local:print-word( ("hello", "world") )
(:
   evaluates the print-word function twice, once for "hello"
   and once for "world", returning hello world
:)
```

Function mapping also works on multiple singleton parameters, resulting in the cross product of all the values (equivalent to nested `for` clauses). In the case of multiple mappings, they occur left to right. For example, the following is evaluated like a nested `for` loop:

```
xquery version "1.0-ml";
(1 to 2) * (3 to 4)
(: returns the sequence (3, 4, 6, 8) :)
```

One consequence of function mapping, which can be surprising the first time you see it, is that if the value passed for a parameter is the empty sequence, it could result in the function being called 0 times (that is, in the function never runs and results in the empty sequence. For example, if you entered the empty sequence as the parameter to the above function call, it returns empty, as follows:

```
xquery version "1.0-ml";

declare function local:print-word ($word as xs:string) { $word };

local:print-word( () )
(:
   evaluates the print-word function zero times, resulting
   in the empty sequence
:)
```

The `local:print-word` function is never called in this case, because it is iterating over the empty sequence, which causes zero invocations of the function. If your function calls are fed by code that can return the empty sequence (an XPath expression, for example), then you might see this behavior.

3.2.2 Enabling or Disabling Function Mapping

In `1.0-ml`, function mapping is enabled by default. In 1.0, it is disabled by default. You can enable it in `1.0` by adding the following to the XQuery prolog:

```
declare namespace xdmp="http://marklogic.com/xdmp";
declare option xdmp:mapping "true";
```

Similarly, you can explicitly disable function mapping in `1.0-ml` by adding the following to the prolog:

```
declare option xdmp:mapping "false";
```

You cannot use function mapping in the `0.9-ml` dialect; if you run code expecting it to map singletons to a sequence in `0.9-ml` (or in `1.0` or `1.0-ml` if function mapping is diabled), it will throw an exception because the sequence cannot be cast to a single string.

3.3 Semi-Colon as Transaction Separator

In the enhanced dialect, you can add a semi-colon after one or more XQuery statements in the body of a main module and then add another one or more XQuery statement. The two sets of statements are then evaluated as two separate transactions. Each set of statements must be a main module; that is, they must all have their own prolog elements. All of the statements in the program must use the same XQuery dialect. For example, the following creates a document and then returns the contents of the document:

```
xquery version "1.0-ml";
xdmp:document-insert("/mydocs/sample.xml",
            <some-element>content</some-element>) ;

xquery version "1.0-ml";
(: Note that the XQuery version must be the same for all
   statements in the module :)
fn:doc("/mydocs/sample.xml")
(: returns the document created in the previous statement :)
```

Note that you cannot use the semi-colon as a transaction separator in the strict XQuery dialect (`1.0`). For more details on transactions, see Understanding Transactions in MarkLogic Server chapter in the *Developer's Guide*.

3.4 Private Function and Variable Definitions

In the `1.0-ml` enhanced dialect, you can create library modules with functions and variables that are private to the library module. Private functions and variables are useful when you have certain code you do not want to expose to users of the library, but might be useful for functions for the library to use. To make functions and variables private, add `private` to the function or variable declaration syntax as follows:

```
declare private function ....

declare private variable ....
```

Note that functions and variables in a main module are private by definition, so declaring them private only makes sense for library modules.

3.5 Functions With Side Effects

The XQuery specification defines that XQuery programs produce only their return values, without producing any side effects; that is, without causing any changes to the run-time environment as a result of running the program (with the exception of `fn:trace`). MarkLogic Server has many enhancements that cause side effects. For example, there are functions that insert or update documents in a database. Since functions like the ones that update documents do more than functions that simply return values, they are extensions to the XQuery specification.

Side effects are extremely useful when building applications. Therefore, MarkLogic Server includes many functions that have side effects. The following are some examples of functions with side effects:

- `xdmp:set`
- Update Built-ins (`xdmp:document-load`, `xdmp:node-insert`, and so on)
- Administrative functions (`xdmp:merge`, Admin library, `xdmp:shutdown`, and so on)

3.6 Shorthand Positional Predicate Syntax

MarkLogic Server enhanced mode supports the shorthand version of the positional predicate syntax, where you can specify the position numbers to include. For example, the following specifies the first three items in the sequence:

```
xquery version "1.0-ml";
(1, 2, 3, 4, 5, 5)[1 to 3]
```

In XQuery 1.0 strict mode (`1.0`), you must use the `fn:position()` function as in the following example:

```
xquery version "1.0";
(1, 2, 3, 4, 5, 5)[fn:position() = (1 to 3)]
```

3.7 Binary Node Constructor and Node Test

MarkLogic Server enhanced mode extends the XQuery types to include a binary node type. Binary nodes are used to store binary documents. To support this type, the MarkLogic Server enhanced XQuery dialect includes a node constructor (`binary`) to construct a binary node and a node test (`binary()`) to test whether a node is a binary node (for example, in a `typeswitch` expression). These extensions are not available in the `1.0` dialect.

3.8 Implementation-Defined Semantics

The XQuery specification lists a number of items that are allowed to be defined by each implementation of XQuery:

http://www.w3.org/TR/xquery/#id-impl-defined-items

This section describes the following implementation-defined items as they are implemented in MarkLogic Server:

- Automatic Namespace Imports for Predefined Namespaces
- External Variables
- Collations
- Implementation-Defined Primitive XQuery Types
- Decimal Precision at Least 18 Digits, and is Not Configurable
- Library Modules Default Function Namespace Defaults to Library Namespace

Note: Except where noted, the items in this section apply all of the XQuery dialects supported in MarkLogic Server.

3.8.1 Automatic Namespace Imports for Predefined Namespaces

Each dialect has a set of namespace prefixes that are predefined. For those predefined namespaces, it is not necessary to declare the prefix. For example, the `fn` prefix is predefined in all of the dialects. For a list of predefined namespaces for each dialect, see "Predefined Namespace Prefixes for Each Dialect" on page 48.

Note: The `fn:` prefix is bound to a different namespace in `1.0` and `1.0-ml` than in `0.9-ml`.

3.8.2 External Variables

External variables are one of the things that the XQuery standard refers to as implementation-defined. In MarkLogic Server, external variables are implemented such that you can pass nodes and values into an XQuery program. To use external variables, you pass in external variables to the XQuery program (via `xdmp:invoke`, `xdmp:eval`, `xdmp:spawn`, or via XCC). The variables are passed in as pairs of QNames and values.

An XQuery program that accepts external variables must declare the external variables in its prolog, as in the following code snippet:

```
declare variable $my:variable as xs:string* external;
```

An XQuery program with this variable declaration would be able to use the string values passed into it via an external variable with the QName `my:variable` (where the namespace prefix `my` was declared somewhere in both the calling and called environments). You could then reference this variable in the XQuery program as in the following example:

```
xquery version "1.0-ml";
declare namespace my="myNamespace";
declare variable $my:variable as xs:string* external;

fn:concat("The value of $my:variable is: ", $my:variable)
```

If you then call this module as follows (assuming the module can be resolved from the path `/extvar.xqy`.

```
xquery version "1.0-ml";
declare namespace my="myNamespace";

xdmp:invoke("/extvar.xqy", (xs:QName("my:variable"), "my value"))
```

This example returns the following string:

```
The value of $my:variable is: my value
```

3.8.3 Collations

The XQuery specification allows collation names and default collation values to be determined by the implementation. MarkLogic Server uses collations to specify the sort order of strings, and it defines the URIs for the collations. Each query runs with a default collation, and that collation can come from the environment (each App Server has a default collation setting) or it can be specified in the XQuery program. Also, you can specify collations for string range indexes and for word lexicons to specify their sort order. For details about collations in MarkLogic Server, including the valid URIs for collations, see Encodings and Collations in the *Developer's Guide*.

3.8.4 Implementation-Defined Primitive XQuery Types

MarkLogic Server has extended the XQuery type system and added some primitive types. These types allow functions to operate on them and are very useful for programming. These types are not required by the XQuery specification, but neither are they in conflict with it because the specification allows implementation-specific primitive types. Therefore, these types are available in all of the XQuery dialects in MarkLogic Server (although in `1.0`, you need to import the namespace prefixes). The following are some of the built-in types in MarkLogic Server:

- `cts:query` (with many subtypes such as `cts:word-query`, `cts:element-query`, and so on)
- `map:map`
- `cts:region` (with subtypes `cts:box`, `cts:circle`, `cts:polygon`, and `cts:point`)

3.8.5 Decimal Precision at Least 18 Digits, and is Not Configurable

MarkLogic Server does not include a facility to limit the maximum precision of a decimal. A decimal has a precision of at least 18 decimal digits (64-bits unsigned). For details, see the XML Schema specification (http://www.w3.org/TR/xmlschema-2/#decimal).

3.8.6 Library Modules Default Function Namespace Defaults to Library Namespace

The default function namespace of an XQuery library module is the namespace of the library module. This allows you to declare functions in the library namespace without prefixing the functions. You can override the default function namespace with a `declare default function namespace` declaration in the prolog of the library module. For library modules where you do not override the default function namespace (and as a general best-practice), you should prefix the XQuery-standard functions (functions with the `fn:` prefix, which is bound to the `http://www.w3.org/2005/xpath-functions` namespace) with the `fn:` prefix. Note that main modules default function namespace defaults to the `fn:` namespace, which is different from library modules.

4.0 XQuery Language

The chapter describes selected parts of the XQuery language. It is not a complete language reference, but it touches on many of the widely used language constructs. For complete details on the language and complete syntax for the language, see the W3C XQuery specification (http://www.w3.org/TR/xquery/). Additionally, there are many third-party books available on the XQuery language which can help with the basics of the language. This chapter has the following sections:

- Expressions Return Items
- XML and XQuery
- XQuery Modules
- XQuery Prolog
- XQuery Comments
- XQuery Expressions
- XQuery Comparison Operators

> **Note:** This chapter describes a subset of the XQuery 1.0 recommendation syntax, which is used in the `1.0` and `1.0-ml` dialects. The syntax for the `0.9-ml` dialect (3.2 compatible) is similar, but not identical to what is described here; most of the differences are in the XQuery prolog. For an overview of the different XQuery dialects, see "XQuery Dialects in MarkLogic Server" on page 7.

4.1 Expressions Return Items

The fundamental building block in XQuery is the *XQuery expression*, which is what the XQuery specification refers to as one or more *ExprSingle* expressions. Each XQuery expression returns a sequence of items; that is, it returns zero or more items, each of which can be anything returned by XQuery (for example, a string, a node, a numeric value, and so on).

Any valid XQuery expression is a valid XQuery. For example, the following is a valid XQuery:

```
"Hello World"
```

It returns the string `Hello World`. It is a simple string literal, and is a valid XQuery. You can combine expressions together using the concatenation operator, which is a comma (`,`), as follows:

```
"Hello", "World"
```

This expression also returns a sequnce of two string `Hello` and `World`. It is two expressions, each returning a single item (therefore it returns a sequence of two strings). In some contexts (a browser, for example), the two strings will be concatenated together into the string `Hello World`.

Expressions can also return no items (the empty sequence), or they can return sequences of items. The following adds a third expression:

```
"Hello", "World", 1 to 10
```

This expression returns the sequence `Hello World 1 2 3 4 5 6 7 8 9 10`, where the sequence 1 to 10 is a sequence of numeric values. You can create arbitrarily complex expressions in XQuery, and they will always return zero or more items.

4.2 XML and XQuery

XQuery is designed for working with XML, and there are several ways to construct and return XML from XQuery expressions. This section describes some of the basic ways to combine XML and XQuery, and contains the following parts:

- Direct Element Constructors: Switching Between XQuery and XML Using Curly Braces
- Computed Element and Attribute Constructors
- Returning XML From an XQuery Program

4.2.1 Direct Element Constructors: Switching Between XQuery and XML Using Curly Braces

As described in the previous section, an XQuery expression by itself is a valid XQuery program. You can create XML nodes as XQuery expressions. Therefore, the following is valid XQuery:

```
<my-element>content goes here</my-element>
```

It simply returns the element. The XQuery syntax also allows you to embed XQuery between XML, effectively “switching” back and forth between an XML syntax and an XQuery syntax to populate parts of the XML content. The separator characters to “switch” back and forth between XML and XQuery are the open curly brace (`{`) and close curly brace (`}`) characters. For example, consider the following XQuery:

```
<my-element>{fn:current-date()}</my-element>
```

This expression returns an XML element named `my-element` with content that is the result of evaluating the expression between the curly braces. This expression returns the current date, so you get an element that looks like the following:

```
<my-element>2008-06-25-07:00</my-element>
```

You can create complex expressions that go “back and forth” between XML and XQuery as often as is needed. For example, the following is slightly more complex:

```
<my-element id="{xdmp:random()}">{fn:current-date()}</my-element>
```

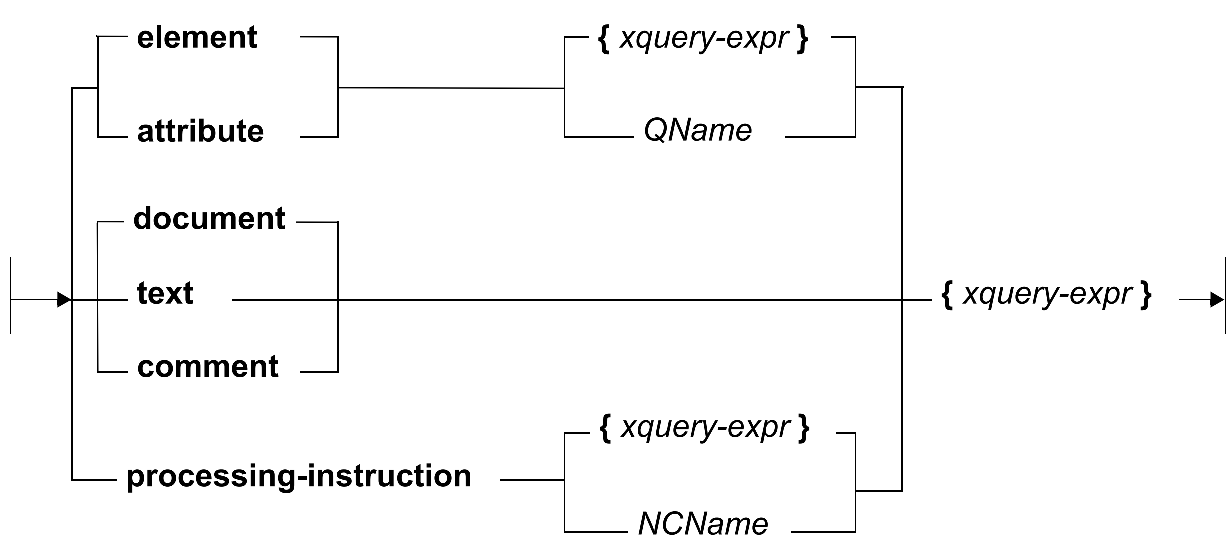
This returns an element like the following:

```
<my-element id="9175848626240925436">2008-06-25-07:00</my-element>
```

This technique of constructing XML are called direct element constructors. There are many more rules for how to use these direct element constructors to create XML nodes. For more details, see the of the XQuery specification (http://www.w3.org/TR/xquery/#doc-xquery-DirCommentConstructor).

4.2.2 Computed Element and Attribute Constructors

You can also create XML nodes by using computed constructors. There are computed constructors for all types of XML nodes (element, attribute, document, text, comment, and processing instruction). The following is the basic syntax for computed constructors:



The following is an example of some XML that is created using computed constructors:

```
element hello { attribute myatt { "world" } , "hello world" }
(:
   returns the following XML:
   <hello myatt="world">hello world</hello>
:)
```

In this example, the comma operator concatenates a constructed attribute (the `myatt` attribute on the `hello` element) and a literal expression (`hello world`, which becomes the element text node content) to create the content for the element node.

4.2.3 Returning XML From an XQuery Program

Using the direct and computed constructors described above, it is natural to have the output of an XQuery program be XML. Besides computed and direct constructors, XML can be the result of an XPath expression, a `cts:search` expression, or any other expression that returns XML. The XML can be constructed as any well-formed XML.

When you construct XML in XQuery, the XQuery evaluator will always construct well-formed XML (assuming your XQuery is valid). Compared with other languages where you construct strings that represent XML, the fact that the XQuery rules ensure that an XML node is well formed tends to eliminate a whole class of bugs in your code that you might encounter using other languages.

4.3 XQuery Modules

While expressions are the building blocks of XQuery coding, *modules* are the building blocks of XQuery programs. There are two kinds of XQuery modules: *main modules* and *library modules*. This section describes XQuery modules and includes the following sections:

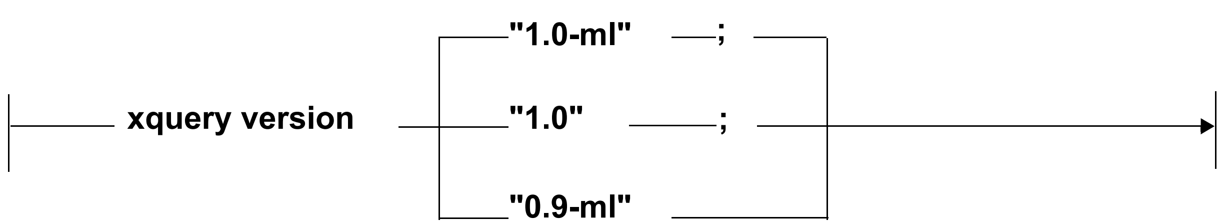
- XQuery Version Declaration
- Main Modules
- Library Modules

This section provides some basic syntax for XQuery modules. For the complete syntax of XQuery modules, see the XQuery specification (http://www.w3.org/TR/xquery/#doc-xquery-Module).

4.3.1 XQuery Version Declaration

Every XQuery module (both main and library) can have an optional XQuery version declaration. The version declaration tells MarkLogic Server which dialect of XQuery to use. MarkLogic Server supports three values for the XQuery version declaration: `1.0-ml`, `1.0`, and `0.9-ml`. For details on the three dialects, including rules for the combining different dialects, see “XQuery Dialects in MarkLogic Server” on page 7.

The following is the basic syntax of the XQuery version declaration:



The following is an example of an XQuery version declaration:

```
xquery version "1.0-ml";
```

4.3.2 Main Modules

A main module contains an XQuery program to be evaluated. You can call a main module directly and it will return the results of the evaluation. A main module contains an optional XQuery version declaration, a prolog (the prolog can be empty, so it is in effect optional), and a body. The XQuery body can be any XQuery expression.

In `1.0-ml`, you can construct programs that have multiple main modules separated by semi-colons, as described in “Semi-Colon as Transaction Separator” on page 16.

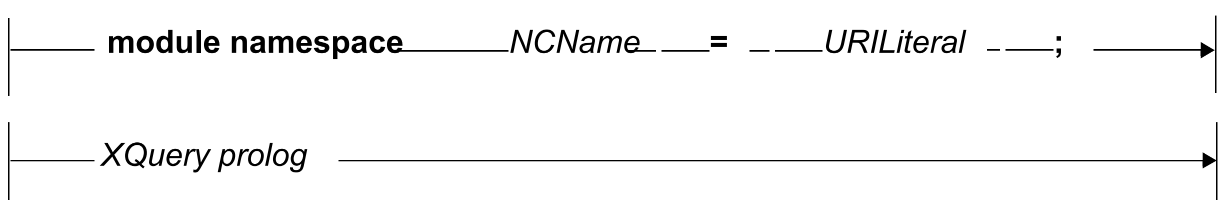
The following is an example of a very simple main module:

```
xquery version "1.0-ml";
"hello world"
```

For another example of a main module, see the example at the end of the “Library Modules” on page 25.

4.3.3 Library Modules

A library module contains function definitions and/or variable definitions. You cannot call a library module to directly evaluate it, and it cannot have a query body. To use a library module, it must be imported from another module (main or library). A library module contains a module declaration followed by a prolog. For details on the prolog, see “XQuery Prolog” on page 26. The following is the basic syntax of a library module:



The following is a very simple XQuery library module

```
xquery version "1.0-ml";
module namespace my-library="my.library.uri" ;

declare function hello() { "hello" };
```

If you stored this module under an App Server root as `hello.xqy`, you could call this function with the following very simple main module:

```
xquery version "1.0-ml";
import module namespace my-library="my.library.uri" at "hello.xqy";

my-library:hello()
     (: this returns the string "hello" :)
```

4.4 XQuery Prolog

The XQuery prolog contains any module imports, namespace declarations, function definitions, and variable definitions for a module. You can have a prolog in either a main module or a library module. The prolog is optional, as you can write an XQuery program with no prolog. This section briefly describes the following parts of the XQuery prolog:

- Importing Modules or Schemas
- Declaring Namespaces
- Declaring Functions
- Declaring Variables
- Declaring a Default Collation

4.4.1 Importing Modules or Schemas

You can import modules and schemas in the XQuery prolog. The following are sample module and schema import declarations:

```
import module namespace my-library="my.library.uri" at "hello.xqy";

import schema namespace xhtml="http://www.w3.org/1999/xhtml"
   at "xhtml1.1.xsd";
```

The library module location and the schema location are not technically required. The location must be supplied for module imports, however, as they are used to determine the location of the library module and the module will not be found without it. Also, all modules for a given namespace should be imported with a single import statement (with comma-separated locations). For schema imports, if the location is not supplied, MarkLogic Server resolves the schema URI using the in-scope schemas (schemas in the schemas database and the `<marklogic-dir>/Config` directory). If there are multiple schemas with the same URI, MarkLogic Server chooses one of them. Therefore, to ensure you are importing the correct schema, you should use the location for the schema import, too. For details on the rules for resolving the locations, see Importing XQuery Modules and Resolving Paths in the *Developer's Guide*.

For more details on imports, see the XQuery specification for schema imports (http://www.w3.org/TR/xquery/#id-schema-import) and for module imports (http://www.w3.org/TR/xquery/#id-module-import).

4.4.2 Declaring Namespaces

Namespace declarations are used to bind a namespace prefix to a namespace URI. The following is a sample namespace declaration:

```
declare namespace my-namespace="my.namespace.uri";
```

For more details on namespace declarations, see the XQuery specification (http://www.w3.org/TR/xquery/#id-namespace-declaration)

4.4.3 Declaring Functions

Functions are a fundamental part of programming in XQuery. Functions provide more than a mechanism to modularize your code (although they certainly are that), as functions allow you to easily perform recursive actions. This is a powerful design pattern in XQuery.

Functions can optionally be typed, both for parmeters to the function and for results of the function. The following is a very simple function declaration that takes a string as input and returns a sentence indicating the length of the string:

```
declare function simple($input as xs:string) as xs:string* {
   fn:concat('The string "', $input, '" is ',
             (fn:string-length($input)),
             ' characters in length.')
} ;
```

4.4.4 Declaring Variables

You can declare variables in a main or library module to reference elsewhere in your programs. If you put variable definitions in a library module, you can reference those variables from any module that imports the library module. Because the content of a variable can be any valid XQuery expression, you can create variables with dynamic content. The following is a variable declaration that returns a string indicating if it is January or not:

```
declare variable $is-it-january as xs:string :=
  if ( fn:month-from-date(fn:current-date()) eq 1 )
  then "it is January"
  else "it is not January" ;
```

If this variable were defined in a library module named `mylib.xqy` stored under your App Server root, and if you imported that library module bound to the namespace prefix `mylib` into a main module, then you can reference this variable in the main module as follows:

```
xquery version "1.0-ml";
import module namespace mylib="my.library.uri" at "mylib.xqy";

$mylib:is-it-january
```

4.4.5 Declaring a Default Collation

The default collation declaration defines the collation that is in effect for a query. In general, everything that uses a collation in a query with a default collation declaration will use the collation specified. The exceptions are for functions that have options which explicitly override the default collation, and for FLWOR expressions that explicitly state the collation in the order by clause. The following is a sample collation declaration:

```
declare default collation "http://marklogic.com/collation/";
```

For more details on collations, see the Encodings and Collations chapter of the *Developer's Guide*.

4.5 XQuery Comments

You can add comments throughout an XQuery program. Comments are surrounded by “smiley face” symbols. The open parenthesis followed by the colon characters (`(:`) denote the start of a comment, and the colon followed by a close parenthesis characters (`(:`) denote the end of a comment. Comments can be nested within comments, which is useful when cutting and pasting code with comments in it into a comment. The following is an example of an XQuery that starts with a comment:

```
(: everything between the smiley faces is a comment :)
"some XQuery goes here"
```

Note: You cannot put a comment inside of a text literal or inside of element content. For example, the following is *not* interpreted as having a comment:

```
<node>(: not a comment :)</node>
```

4.6 XQuery Expressions

This section describes the following XQuery expressions:

- XPath Expressions
- FLWOR Expressions
- The typeswitch Expression
- The if Expression
- Quantified Expressions (some/every ... satisfies ...)
- Validate Expression

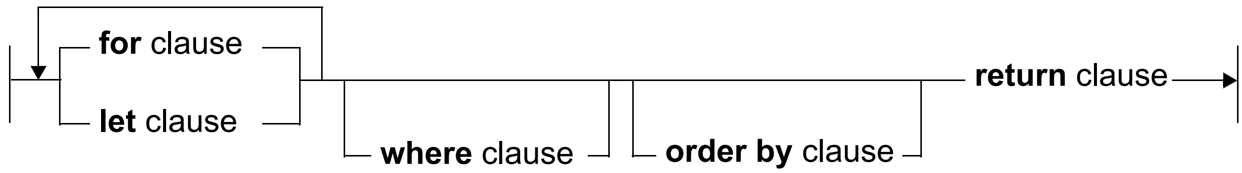
4.6.1 XPath Expressions

XPath expressions search for XML content. They can be combined with other XQuery expressions to form other arbitrarily complex expressions. For more details on XPath expressions, see “XPath Quick Reference” on page 40.

4.6.2 FLWOR Expressions

The FLWOR expression (`for`, `let`, `where`, `order by`, `return`) is used to generate items or sequences. A FLWOR expression binds variables, applies a predicate, orders the data set, and constructs a new result:

The following is the basic syntax of a FLWOR expression:

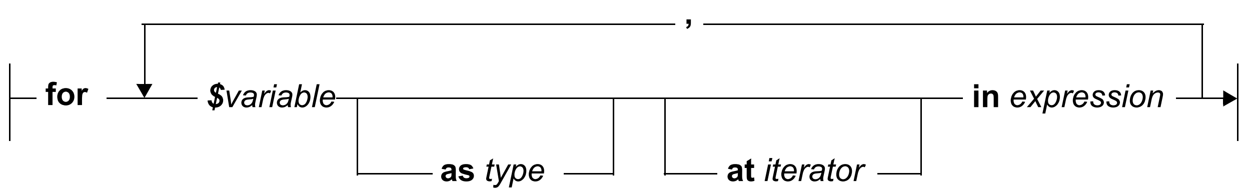


The following sections examine each of the five clauses in more detail:

- The for Clause
- The let Clause
- The where Clause
- The order by Clause
- The return Clause

4.6.2.1 The for Clause

The `for` clause is used for iterating over one or more sequences:



The `for` clause iterates over each item in the expression to which the variable is bound. In the `return` clause, an action is typically performed on each item in the variable bound to the expression. For example, the following binds a sequence of integers to a variable and then performs an action (multiplies it by 2) on each item in the sequence:

```
for $x in (1, 2, 3, 4, 5)
return
$x * 2

(: returns the sequence (2, 4, 6, 8, 10) :)
```

As is common in XQuery, order is significant, and the items are bound to the variable in the order they are output from the expression.

You can also bind multiple variables in one or more `for` clauses. The FLWOR expression then iterates over each item in the subsequent variables once for each item in the first variable. For example:

```
for $x in (1,2,3)
for $y in (4,5,6)
return
$x * 2

(: returns the sequence (2, 2, 2, 4, 4, 4, 6, 6, 6) :)
```

In this case, the inner `for` loop (with `$y`) is executed one complete iteration for each of the items in the outer `for` loop (the one with `$x`). Even though it does not return anything from $y, the expression in the return clause is evaluated once for each item in `$y`, and that happens once for each item in `$x`.

You could return something from each iteration of $y, as in the following example:

```
for $x in (1,2,3)
for $y in (4,5,6)
return
($x * 2, $y * 3)

(: returns the sequence
(2, 12, 2, 15, 2, 18, 4, 12, 4, 15, 4, 18, 6, 12, 6, 15, 6, 18) :)
```

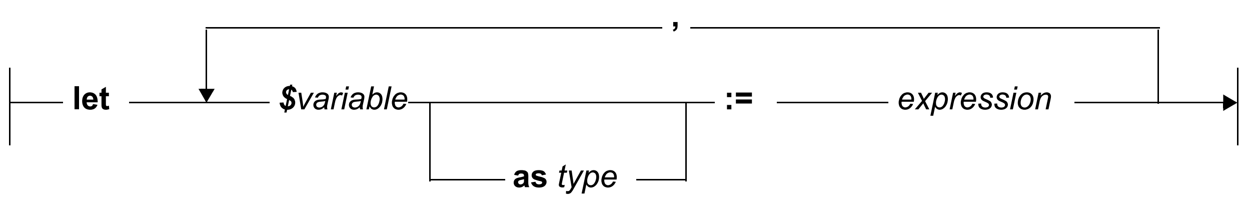
Alternately, you could write the two for clauses as follows, with the same results:

```
for $x in (1,2,3), $y in (4,5,6)
```

When you have multiple variables bound in `for` clauses, it is an effective way of joining content from one variable with the other. Note that if the content from each variable comes from a different document, then multiple `for` clauses in a FLOWR expression ends up performing a join of the documents.

4.6.2.2 The let Clause

The `let` clause is used for binding variables (without iteration) to a single value or to sequences of values:



A `let` clause produces a single binding for each variable. Consequently, `let` clauses do not affect the number of binding tuples evaluated in a FLWOR expression. Variables bound in a `let` clause are available to anything that follows in the FLWOR expression (for example, subsequent `for` or `let` clauses, the `where` clause, the `order by` clause, or the `return` clause).

In its simplest form, the let clause allows you to build a FLWOR expression that outputs the sequence to which the variable is bound. For example, the following expression:

```
let $seq := ("hello", "goodbye") return $seq
```

is equivalent to the following expression:

```
"hello", "goodbye"
```

They each return the two item sequence `hello goodbye`.

A typical use for a let clause is to bind a sequence to a variable, then use the variable in a for clause to iterate over each item. For example:

```
let $x := (1 to 5)
for $y in $x
return
$x * 2

(: returns the sequence (2, 4, 6, 8, 10) :)
```

Again, this is a trivial example, but it could be that the expression in the `let` binding is complicated, and this technique allows you to cleanly structure your code.

4.6.2.3 The where Clause

The `where` clause specifies a filter condition on the tuples emerging from the `for-let` portion of a FLWOR expression:

Only tuples for which the *boolean-expression* evaluates to *true* will contribute to the result sequence of the FLWOR expression The where clause preserves the order of tuples, if any. *boolean-expression* may contain `and`, `or` and `not`, among other operators.

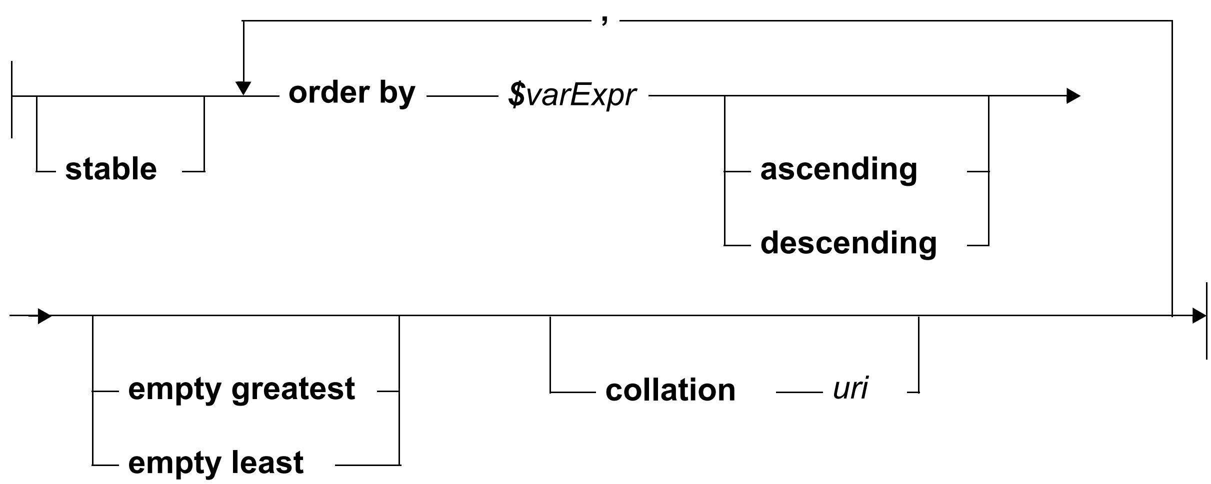
Typically, you use comparison operators to test for some condition in a `where` clause. For example, if you only want to output from the FLWOR items that start with the letter “a”, you can do something like the following:

```
for $x in ("a", "B", "c", "A", "apple")
where fn:starts-with(fn:lower-case($x), "a")
return
$x

(: returns the sequence ("a", "A", "apple") :)
```

4.6.2.4 The order by Clause

The `order by` clause specifies the order (ascending or descending) to sort items returned from a FLWOR expression, and also provides an option to specify a collation URI with which to determine the order:



The `order by` clause can be used to specify an order in which the tuple sequence will be passed to the `return` clause. The `order by` clause can specify any sort key, regardless of whether that sort key is contained in the result sequence. You can reorder sequences on an ascending or descending basis. The following sorts the sequence bound to `$x` (in collation order) by each item:

```
for $x in ( "B", "c", "a", "d")
order by $x
return $x (: returns the sequence ("a", "B", "c", "d") :)
```

4.6.2.5 The return Clause

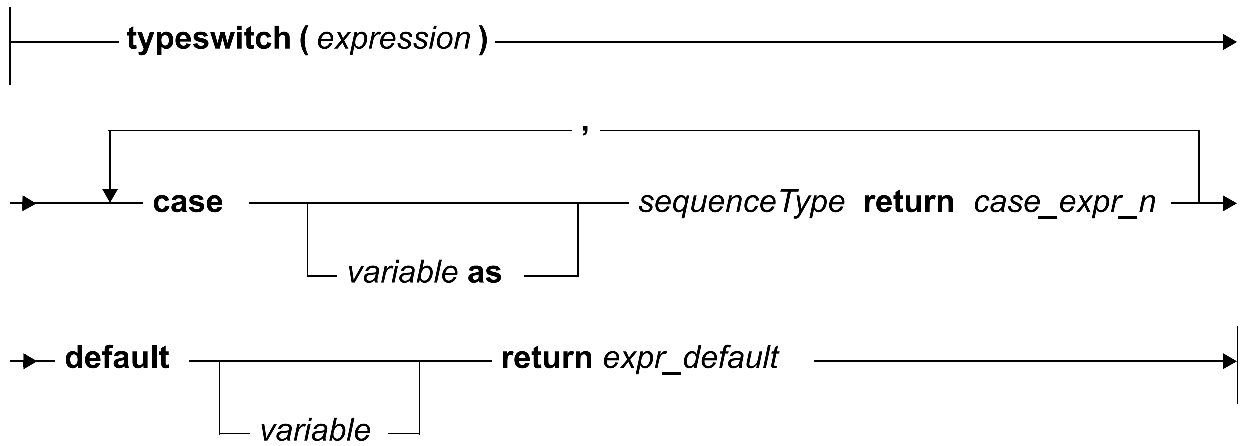
The `return` clause constructs the result of a FLWOR expression:

The return expression is evaluated once for each tuple of bound variables. This evaluation preserves the order of tuples, if any, or it can impose a new order using the `order by` clause.

Because the return `clause` specifies an expression, any legal XQuery expression can be used to construct the result, including another FLWOR expression.

4.6.3 The typeswitch Expression

The `typeswitch` expression allows conditional evaluation of a set of sub-expressions based on the type of a specified expression:



A `typeswitch` expression evaluates the first *case_expr* whose *sequenceType* matches the type of the specified *expression*. If there is no *sequenceType* match, *expr_default* is evaluated.

Typeswitch provides a powerful mechanism for processing node contents:

```
typeswitch ($address)
  case $a as element(*, USAddress) return handleUS($a)
  case $a as element(*, CanadaAddress) return handleCanada($a)
  default return handleUnknown($address)
```

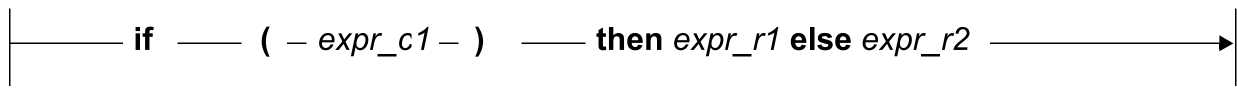
This code snippet determines the *sequenceType* of the variable `$address`, then evaluates one of three sub-expressions. In this case:

- If `$address` is of type `USAddress`, the function `handleUS($a)` is evaluated.
- If `$address` is of type `CanadaAddress`, the function `handleCanada($a)` is evaluated.
- If the type of variable `$address` matches none of the above, the function `handleUnknown($a)` is evaluated.

A *sequenceType* can also be a kind test (such as an element test). It is possible to construct `case` clauses in which a particular *expression* matches multiple *sequenceTypes*. In this case, the *case_expr* of only the first matching *sequenceType* is evaluated. You can also use the typeswitch expression in a recursive function to iterate through a document and perform transformation of the document. For details about using recursive typeswitches, see the Transforming XML Structures With a Recursive typeswitch Expression chapter of the *Developer's Guide*.

4.6.4 The if Expression

The `if` expression allows conditional evaluation of sub-expressions:



If expression *expr_c1* evaluates to true, then the value of the `if` expression is the value of expression *expr_r1*, otherwise the value of the `if` expression is the value of *expr_r2*. The `else` clause is not optional; if no action is to be taken, an empty sequence should be used for *expr_r2*; there is no “end if” or similar construct in XQuery:

```
if (1 eq 2)
then "this is strange"
else ()
```

The extent of *expr_r1* and *expr_r2* is limited to a single expression. If a more complex set of actions are required, an element constructor, sequence, or function call must be used.

`If` expressions can be nested:

```
if ($year < 1994)
then
   <available>archive</available>
else if ($year = $current_year) then
   <available>current</available>
else
   <available>inventory</available>
```

4.6.5 Quantified Expressions (some/every ... satisfies ...)

XQuery provides predicates that simplify the evaluation of quantified expressions. The basic syntax for these expressions follows:

some *var* **in** *expr* **satisfies** *predicate*

every *var* **in** *expr* **satisfies** *predicate*

These expressions are particularly useful when trying to select a node based on a condition satisfied by at least one or alternatively all of a particular set of its children.

Imagine an XML document containing log messages. The document has the following structure:

```
<log>
   <event>
      <program>   .... </program>
      <message>   .... </message>
      <level>     .... </level>
      <error>
         <code>      .... </code>
         <severity>  .... </severity>
         <resolved>  .... </resolved>
      </error>
      <error>
         ....
      </error>
      ....
   </event>
   ....
</log>
```

Every <event> node has <program>, <message>, and <level> children. Some <event> nodes have one or more <error> children.

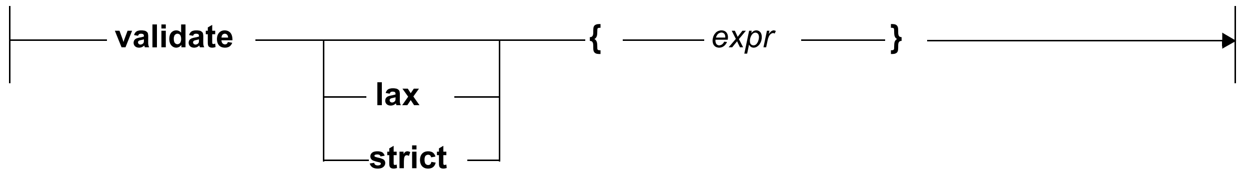
Consider a query to report on those events that have unresolved errors:

```
for $event in /log/event
where some $error in $event/error satisfies $error/resolved = "false"
return
   $event
```

This query returns only those <event> nodes in which there is an <error> node with a <resolved> element whose value is “false”.

4.6.6 Validate Expression

The validate expression is used to validate element nodes against in-scope schemas (schemas that are in the schemas database). The following is the basic syntax of the validate expression:



The expression to validate should be a simple XML node. The node can reference a schema. For more details, see the XQuery specification (http://www.w3.org/TR/xquery/#id-validate).

> **Note:** MarkLogic Server only validates simple XML types. Complex XML types are only partially validated.

The following is a simple validate expression:

```
xquery version "1.0-ml";
validate { <p xmlns="http://www.w3.org/1999/xhtml">hello there</p> }

(:
validates against the in-scope xhtml schema and returns the element:
<p xmlns="http://www.w3.org/1999/xhtml">hello there</p>
:)
```

4.7 XQuery Comparison Operators

This section lists the comparison operators in XQuery. The purpose of the operators are to compare expressions. This section includes the following parts:

- Node Comparison Operators
- Sequence and Item Operators

4.7.1 Node Comparison Operators

You can specify node comparisons to test if two nodes are before or after each other (in document order), or if the nodes are the exact same node. These tests return true or false. The following are the node comparison operators:

<table>
<tr><th>Operator</th><th>Description</th><th>Example</th></tr>
<tr><td><<</td><td>The node before operator. Tests if a node comes before another node in document order.</td><td><pre>let $x := <foo>
            <bar>hello</bar>
            <baz>goodbye</baz>
          </foo>
return
($x/baz << $x/bar, $x/bar << $x/baz)
(: returns false, true :)</pre></td></tr>
<tr><td>>></td><td>The node after operator. Tests if a node comes after another node in document order.</td><td><pre>let $x := <foo>
            <bar>hello</bar>
            <baz>goodbye</baz>
          </foo>
return
($x/baz >> $x/bar, $x/bar >> $x/baz)
(: returns true, false :)</pre></td></tr>
<tr><td>is</td><td>The is operator. Tests if a node is the exact same node as another (does not just test equality).</td><td><pre>let $x := <foo>
            <bar>hello</bar>
            <baz>goodbye</baz>
          </foo>
return
($x/baz is $x/bar, $x/bar is $x/bar)
(: returns false, true :)</pre></td></tr>
</table>

Node comparison tests are useful when creating logic that relies on document order. For example, if you wanted to verify if a particular node came before another node, you can test as follows:

```
$x << $y
```

If this test returns true, you know that the node bound to `$x` comes before the node bound to `$y`, based on document order.

4.7.2 Sequence and Item Operators

XQuery has separate operators for to compare sequences and items.The following tabled lists XQuery operators for sequences and for items, along with a description and example for each operator. These operators are used to form expressions that compare values, and those expressions return a boolean value. This section consists of the following parts:

- Sequence Operators
- Item Operators

4.7.2.1 Sequence Operators

The following operators work on sequences. Note that a single item is a sequence, so the sequence operators can work to compare single items. A sequence operator is true if any of the comparisons are true.

Operator	Description	Example
`=`	The equality operator. Operates on sequences (which can contain 0 or more items).	`1 eq 1 => true` `1 eq (1, 2) => true`
`>`	Greater than operator. Operates on sequences (which can contain 0 or more items).	`1 > 1 => false`
`>=`	Greater than or equal operator. Operates on sequences (which can contain 0 or more items).	`1 >= 1 => true` `1 >= (1, 2) => true`
`<`	Less than operator. Operates on sequences (which can contain 0 or more items). Returns true if any item in the sequence is greater.	`1 < 1 => false` `1 < (1, 2) => true`
`<=`	Less than or equal operator. Operates on sequences (which can contain 0 or more items). Returns true if any item in the sequence is less than or equal to.	`1 <= 1 => true` `1 <= (1, 2) => true`
`!=`	The not equal operator. Operates on sequences (which can contain 0 or more items). Returns true if any item in the sequence is not equal	`1 != (1, 2) => true`

4.7.2.2 Item Operators

The following operators work on items. If you use these operators on a sequence, in the `1.0-ml` dialect they will perform function mapping, and the value will be the effective boolean value of the sequence of results. In 1.0 and 0.9-ml, they will throw an `XDMP-MANYITEMSEQ` exception if you try to compare a sequence of more than one item.

Operator	Description	Example
`eq`	The equality operator. Operates only on single items.	`1 eq 1 => true` `1 eq (1, 2) => error`
`gt`	Greater than operator. Operates only on single items.	`1 gt 1 => false`
`ge`	Greater than or equal operator. Operates only on single items.	`1 ge 1 => true`
`lt`	Less than operator. Operates only on single items.	`1 lt 1 => false`
`le`	Less than or equal operator. Operates only on single items.	`1 le 1 => true`
`ne`	The not equal operator. Operates on single items.	`1 ne 1 => false`

5.0 XPath Quick Reference

The section provides a brief overview of the basics of XPath, and includes the following sections:

- Path Expressions
- XPath Axes and Syntax
- XPath 2.0 Functions

For detailed information about XPath, see the W3C XPath 2.0 language reference (http://www.w3.org/TR/xpath20/).

5.1 Path Expressions

XPath 2.0 is part of XQuery 1.0. XPath is used to navigate XML structures. In MarkLogic Server, the XML structures can be stored in a database or they can be constructed in XQuery. A *path expression* is an expression that selects nodes from an XML structure. Path expressions are a fundamental way of identifying content in XQuery. Each path has zero or more *steps*, which typically select XML nodes. Each step can have zero or more *predicates*, which constrain the nodes that are selected. By combining multiple steps and predicates, you can create arbitrarily complex path expressions. Consider the following path expression (which is in itself a valid XQuery expression):

```
//LINE[fn:contains(., "To be, or not to be")]
```

Against the Shakespeare database (the XML is available at http://www.oasis-open.org/cover/bosakShakespeare200.html), this XPath expression selects all `LINE` elements that contain the text `To be or not to be`. You can then walk up the document to its parent to see who says this line. as follows:

```
//LINE[fn:contains(., "To be, or not to be")]/../SPEAKER
```

This returns the following line:

```
<SPEAKER>HAMLET</SPEAKER>
```

You can make path expressions arbitrarily complex, which makes them a very powerful tool for navigating through XML structures. For more details about path expressions, see the W3C XQuery specification (http://www.w3.org/TR/xquery/#id-path-expressions).

A path expression always returns nodes in document order. If you want to return nodes in relevance order (that is, relevance-ranked nodes), use the MarkLogic Server `cts:search` built-in function or put the XPath in a FLWOR expression with an `order by` clause. Note that both XPath expressions and `cts:search` expressions use any available indexes for fast expression evaluation. For details on `cts:search`, see the *Developer's Guide* and the *XQuery Built-In and Module API Documentation*. For details about index options in MarkLogic Server, see the *Administrator's Guide*.

5.2 XPath Axes and Syntax

The following table shows the XPath axes supported in MarkLogic Server.

Axis	Description	Shorthand (N/A if no shorthand)
`ancestor::`	Selects all ancestor nodes, which includes the parent node, the parent's parent node, and so on.	N/A
`ancestor-or-self::`	Selects the current node as well as all ancestor nodes, which includes the parent node, the parent's parent node, and so on.	N/A
`attribute::`	Selects the attributes of the current node.	`@`
`child::`	Selects the immediate child nodes of the current node.	`/`
`descendant::`	Selects all descendant nodes (child nodes, their child nodes, and so on).	N/A
`descendant-or-self::`	Selects the current node as well as all descendant nodes (child nodes, their child nodes, and so on).	`//`
`following::`	Selects everything following the current node.	`>>`
`following-sibling::`	Selects all sibling nodes (nodes at the same level in the XML hierarchy) that come after the current node.	N/A
`namespace::`	Selects the namespace node of the current node.	N/A
`parent::`	Selects the immediate parent of the current node.	`..`
`preceding::`	Selects everything before the current node.	`<<`
`preceding-sibling::`	Selects all sibling nodes (nodes at the same level in the XML hierarchy) that come before the current node.	N/A

Axis	Description	Shorthand (N/A if no shorthand)
`property::`	MarkLogic Server enhancement. Selects the properties fragment corresponding to the current node.	N/A
`self::`	Selects the current node (the context node).	.

Keep in mind the following notes when using the XPath axes:

- XPath expressions are always returned in document order.
- Axes that look forward return in document order (closest to farthest away from the context node).
- Axes that look backward return in reverse document order (closest to farthest away from the context node).
- The context node is the node from which XPath steps are evaluated. The context node is sometimes called the current node.

5.3 XPath 2.0 Functions

The XQuery standard functions are the same as the XPath 2.0 functions. These XQuery-standard functions are all built into MarkLogic Server, and use the namespace bound to the `fn` prefix, which is predefined in MarkLogic Server. For details on these functions, see the *XQuery Built-In and Module API Documentation* reference.

6.0 Understanding XML Namespaces in XQuery

XQuery is designed to work well with XML content, allowing many convenient ways to search through XML elements and attributes as well as making it easy to output XML from an XQuery program. When working with XML, you must understand a little about the XML data model, and one fundamental aspect of the XML data model is namespaces. This chapter describes XML namespaces and how they are important in XQuery, and includes the following sections:

- XML QNames, Local Names, and Namespaces
- Everything Is In a Namespace
- XML Data Model Versus Serialized XML
- Declaring a Default Element Namespace in XQuery
- Tips For Constructing QNames
- Predefined Namespace Prefixes for Each Dialect

6.1 XML QNames, Local Names, and Namespaces

XML uses qualified names, also called *QNames*, to uniquely identify elements and attributes. A QName for an XML element or attribute has two parts: the *namespace name* and the *local name*. Together, the namespace and local name uniquely define how the element or attribute is identified. Additionally, the QName also retains its namespace prefix, if there is one. A namespace prefix binds a namespace URI to a specified string (the string is the prefix).

6.2 Everything Is In a Namespace

In XML and XQuery, element and attribute nodes are always in a namespace, even if that namespace is the *empty namespace* (sometimes called *no namespace*). Each namespace has a uniform resource identifier (URI) associated. A URI is essentially a unique string that identifies the namespace. That string can be bound to a namespace prefix, which is just a shorthand string which is used to identify a (usually longer) namespace name. When something is in the empty namespace, the namespace name is the empty string ("").

There can also be a default element namespace defined for the module, as described in "Declaring a Default Element Namespace in XQuery" on page 47. The fact that every element is in a namespace, along with the fact that XPath expressions of an unknown node return the empty sequence, make it easy to have simple coding errors (or even typographic errors) that cause your query to be a valid XPath expression, but to return the empty string. For example, if you have a simple typographical error in a namespace declaration, then XPath expressions that you might expect to return nodes might return the empty sequence. Consider the following query against a database with XHTML content:

```
xquery version "1.0-ml";
declare namespace xh="http://www.w3.org/1999/html";
//xh:p
```

You might expect this to return all of the XHTML `p` elements in the database, but instead it returns nothing (the empty sequence). If you look closely, though, you will notice that the namespace URI is misspelled (it is missing the `x` in `xhtml`). If you keep in mind that everything is in a namespace, it can help find many simple XQuery coding errors. The correct version of this query is as follows, and will return all of the XHTML `p` elements:

```
xquery version "1.0-ml";
declare namespace xh="http://www.w3.org/1999/xhtml";
//xh:p
```

6.3 XML Data Model Versus Serialized XML

This section highlights the difference between the XML data model, used to programmatically access XML content, and the serialized form of XML, used to display the XML in human-readable form. The following topics are covered:

- XQuery Accesses the XML Data Model
- Serialized XML: Human-Readable With Angle Brackets
- Understanding Namespace Inheritance With the xmlns Attribute

6.3.1 XQuery Accesses the XML Data Model

When an XQuery program accesses XML, it accesses it through the XML data model. The XML data model access nodes via their QNames, which are pairs of namespace name and local name. The XML data model does not store namespace prefixes. You can use namespace prefixes to access XML if those prefixes are in-scope in your XQuery (that is, if the prefixes are bound to a namespace). In-scope prefixes are a combination of any prefixes bound to a namespace in your query and the predefined namespace prefixes defined in “Predefined Namespace Prefixes for Each Dialect” on page 48.

The XML data model is aware of XML schema, and all XML nodes can optionally have XML types (for example, `xs:string`, `xs:dateTime`, `xs:integer`, and so on). When you are creating library functions that might be called from a number of contexts, knowing that XQuery accesses the XML data model can help you to make your code robust. For example, you might have code that explicitly (or implicitly, using the XQuery rules) casts nodes to a particular XML type, enforcing strong typing in your code.

6.3.2 Serialized XML: Human-Readable With Angle Brackets

When XML nodes are transformed from their internal, XML data model representation to a human-readable form, the process is known as *XML serialization*. A serialized XML node contains all of the namespace information, although some namespace prefixes may or may not be included in the serialization. Serialized XML does not generally contain the type information or the schema information; it is up to the XQuery program to specify a schema for a given XML representation.

When serializing XML, there are five XML reserved characters that are serialized with their corresponding XML entities. These characters cannot appear as content in a serialized XML text node. The following table shows these five characters:

Character	XML Entity	Name of Character
"	"	double quotation mark
&	&	ampersand
'	'	apostrophe
<	<	less-than sign
>	>	greater-than sign

There are different ways to serialize the same XML content. The way XML content is serialized depends on how the content is constructed, the various namespace declarations in the query, and how the XML content was loaded into MarkLogic Server (for content loaded into a database). In particular, the ampersand character can be tricky to construct in an XQuery string, as it is an escape character to the XQuery parser. The ways to construct the ampersand character in XQuery are:

- Use the XML entity syntax (for example, &).
- Use a CDATA element (<![CDATA[element content here]]>), which tells the XQuery parser to read the content as character data.
- Use the repair option on xdmp:document-load, xdmp:document-get, or xdmp:unquote.

For example, consider the following query:

```
xquery version "1.0-ml";
declare default element namespace "my.namespace.hello";

<some-element><![CDATA{element content with & goes
              here]]></some-element>
```

If you evaluate this query, it returns the following serialization of the specified element:

```
<some-element xmlns="my.namespace.hello">element content
     with &amp; goes here</some-element>
```

If you consider a similar query with a namespace prefix binding instead of the default element namespace declaration:

```
xquery version "1.0-ml";
declare namespace hello="my.namespace.hello";

<hello:some-element><![CDATA[element content with & goes
                    here]]></hello:some-element>
```

If you evaluate this query, it returns the following serialization of the specified element:

```
<hello:some-element xmlns:hello="my.namespace.hello">element
    content with &amp; goes here</hello:some-element>
```

Notice that in both cases, the `&` character is escaped as an XML entity, and in each case there is an `xmlns` attribute added to the serialization. In the first example, there is no prefix bound to the namespace, but in the second one there is (because it is declared in the query). Both serializations represent the exact same XML data model.

To construct the double quotation mark and apostrophe characters within a string quoted with one of these characters (`'` or `"`), you can use the character to escape itself, or you can quote the string with the other quote character, as follows:

```
"""" (: returns a single character: " :)

'"' (: returns a single character: " :)

'''' (: returns a single character: ' :)

"'" (: returns a single character: ' :)
```

6.3.3 Understanding Namespace Inheritance With the xmlns Attribute

As seen in the previous example, XML has a namespace declaration called `xmlns`, which is used to specify namespaces in XML. An `xmlns` namespace declaration looks like an attribute (although it is not actually an attribute). It can either stand by itself or have a prefix appended to it, separated by a colon (`:`) character. Any `xmlns` namespace declaration is inherited by all of its child elements, and if it has a prefix appended to it, the children also inherit the namespace prefix binding.

For example, the following XML serialization specifies that the XHTML namespace is inherited from the root element:

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <body><p>This is in the XHTML namespace</p></body>
</html>
```

Each of the elements (`html`, `body`, and `p` in this example) are in the XHTML namespace.

Similarly, an `xmlns` namespace declaration with a prefix appended specifies that the prefix is inherited by the element children.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:my="my.namespace">
  <body>
    <p>This is in the XHTML namespace</p>
    <my:p>This element is in my.namespace</my:p>
  </body>
</html>
```

There are some other subtleties of namespace inheritance in XML. For more details, see the XML Schema specification (http://www.w3.org/XML/Schema).

6.4 Declaring a Default Element Namespace in XQuery

An XQuery program can declare a namespace as the default element namespace for any elements that do not have a namespace. By default, the default element namespace is no namespace, which is denoted by the empty string URI (`""`). If you want to define a default element namespace for a query, add a declaration to the prolog similar to the following, which declares the XHTML namespace (`http://www.w3.org/1999/xhtml`) as the default element namespace:

```
declare default element namespace "http://www.w3.org/1999/xhtml";
```

An XQuery program that has this prolog declaration will use the XHTML namespace for all elements where a namespace is not explicitly defined (for example, with a namespace prefix).

Declaring a default element namespace is a convenience and a style which some programmers find useful. While it is sometimes convenient (so you do not have to prefix element names, for example), it can also cause confusion in larger programs that use multiple namespace, so for more complex programming efforts, explicitly defining namespaces is usually more clear.

6.5 Tips For Constructing QNames

In XML, elements and attributes are uniquely identified by a qualified names (QNames, as described in "XML QNames, Local Names, and Namespaces" on page 43). A QName is a pairing of a namespace name and a local name, and it uniquely describes an element or attribute name. XQuery also uses QNames to uniquely identify function names, variable names, and type names.

There are many functions that use QNames in XQuery, and all of the rules for in-scope namespaces apply to constructing those QNames. For example, if the namespace prefix `my` is bound to the namespace URI `my.namespace` in the scope of a query, then the following would construct a QName in that namespace with the local name `some-element`:

```
xs:QName("my:some-element")
```

Similarly, you can construct this QName using the `fn:QName` function as follows:

```
fn:QName("my.namespace", "some-element")
```

Because a prefix is not specified in the second parameter to the above function, the QName is defined to have a prefix of the empty string ("").

Similarly, you can construct this QName with the prefix `my` by using the `fn:QName` function as follows:

```
fn:QName("my.namespace", "my:some-element")
```

XQuery functions and other language constructs that take a QName can use any in-scope namespace prefixes. For example, the following will construct an `html` element in the XHTML namespace:

```
xquery version "1.0-ml";
declare namespace xh="http://www.w3.org/1999/xhtml";

element xh:html { "This is in the xhtml namespace." }
```

6.6 Predefined Namespace Prefixes for Each Dialect

This section lists the namespaces that are predefined for each of the dialects supported in MarkLogic Server. When a prefix is predefined, you can use it in your XQuery without the need to define it in a `declare namespace` prolog statement. It contains the following parts:

- 1.0-ml Predefined Namespaces
- 1.0 Predefined Namespaces
- 0.9-ml Predefined Namespaces

6.6.1 1.0-ml Predefined Namespaces

The following table lists the namespace prefixes and the corresponding URIs to which they are bound that are predefined in the 1.0-ml XQuery dialect.

`1.0-ml` Predefined Prefix	Used For	Namespace URI
`cts`	MarkLogic Server search functions (Core Text Services)	`http://marklogic.com/cts`
`dav`	Used with WebDAV	`DAV:`
`dbg`	Debug Built-In functions	`http://marklogic.com/xdmp/debug`

1.0-ml Predefined Prefix	Used For	Namespace URI
dir	MarkLogic Server directory XML	http://marklogic.com/xdmp/directory
err	namespace for XQuery and XPath errors	http://www.w3.org/2005/xqt-errors
error	MarkLogic Server error namespace	http://marklogic.com/xdmp/error
fn	XQuery standard function namespace	http://www.w3.org/2005/xpath-functions
local	local namespace for functions defined in main modules	http://www.w3.org/2005/xquery-local-functions
lock	MarkLogic Server locks	http://marklogic.com/xdmp/lock
map	MarkLogic Server maps	http://marklogic.com/xdmp/map
math	math Built-In functions	http://marklogic.com/xdmp/math
prof	profile Built-In functions	http://marklogic.com/xdmp/profile
prop	MarkLogic Server properties	http://marklogic.com/xdmp/property
sec	security Built-In functions	http://marklogic.com/xdmp/security
spell	spelling correction functions	http://marklogic.com/xdmp/spell
xdmp	MarkLogic Server Built-In functions	http://marklogic.com/xdmp
xml	XML namespace	http://www.w3.org/XML/1998/namespace
xmlns	xmlns namespace	http://www.w3.org/2000/xmlns/

1.0-ml Predefined Prefix	Used For	Namespace URI
xqe	deprecated MarkLogic Server xqe namespace	http://marklogic.com/xqe
xqterr	XQuery test suite errors (same as err)	http://www.w3.org/2005/xqt-errors
xs	XML Schema namespace	http://www.w3.org/2001/XMLSchema

6.6.2 1.0 Predefined Namespaces

The following table lists the namespace prefixes and the corresponding URIs to which they are bound that are predefined in the 1.0 XQuery dialect (strict XQuery 1.0).

1.0 Predefined Prefix	Used For	Namespace URI
err	namespace for XQuery and XPath errors	http://www.w3.org/2005/xqt-errors
fn	XQuery standard function namespace	http://www.w3.org/2005/xpath-functions
local	local namespace for functions defined in main modules	http://www.w3.org/2005/xquery-local-functions
xml	XML namespace	http://www.w3.org/XML/1998/namespace
xmlns	xmlns namespace	http://www.w3.org/2000/xmlns/
xs	XML Schema namespace	http://www.w3.org/2001/XMLSchema

6.6.3 0.9-ml Predefined Namespaces

The following table lists the namespace prefixes and the corresponding URIs to which they are bound that are predefined in the 0.9-ml XQuery dialect (MarkLogic Server legacy).

`0.9-ml` Predefined Prefix	Used For	Namespace URI
`cts`	MarkLogic Server search functions (Core Text Services)	`http://marklogic.com/cts`
`dav`	Used with WebDAV	`DAV:`
`dbg`	Debug Built-In functions	`http://marklogic.com/xdmp/debug`
`dir`	MarkLogic Server directory XML	`http://marklogic.com/xdmp/directory`
`err`	MarkLogic Server error namespace (note this is different than `1.0` and `1.0-ml`)	`http://marklogic.com/xdmp/error`
`error`	MarkLogic Server error namespace	`http://marklogic.com/xdmp/error`
`fn`	XQuery standard function namespace (not this has a different namespace URi than `1.0` and `1.0-ml`)	`http://www.w3.org/2003/05/xpath-functions`
`lock`	MarkLogic Server locks	`http://marklogic.com/xdmp/lock`
`map`	MarkLogic Server maps	`http://marklogic.com/xdmp/map`
`math`	math Built-In functions	`http://marklogic.com/xdmp/math`
`prof`	profile Built-In functions	`http://marklogic.com/xdmp/profile`

0.9-ml Predefined Prefix	Used For	Namespace URI
prop	MarkLogic Server properties	http://marklogic.com/xdmp/property
sec	security Built-In functions	http://marklogic.com/xdmp/security
spell	spelling correction functions	http://marklogic.com/xdmp/spell
xdt	May 2003 duration namespace	http://www.w3.org/2003/05/xpath-datatypes
xdmp	MarkLogic Server Built-In functions	http://marklogic.com/xdmp
xml	XML namespace	http://www.w3.org/XML/1998/namespace
xmlns	xmlns namespace	http://www.w3.org/2000/xmlns/
xqe	deprecated MarkLogic Server xqe namespace	http://marklogic.com/xqe
xqterr	XQuery test suite errors (same as err)	http://www.w3.org/2005/xqt-errors
xs	XML Schema namespace	http://www.w3.org/2001/XMLSchema

7.0 Application Programming in XQuery

In MarkLogic Server, XQuery is not only used to query XML, but is also used as a programming language to create applications. It is especially powerful as a programming language to create web applications, as you can easily write XQuery code that outputs XHTML, which is the XML variant of HTML. This chapter describes some of the language features that make XQuery particularly useful as an application programming language, and includes the following sections:

- Design Patterns
- Using Functions
- Search Functions
- Updates and Transactions
- HTTP App Server Functions
- Additional Resources

7.1 Design Patterns

For any programming language, there are design patterns that develop over time to perform various tasks. In XQuery with MarkLogic Server, one design pattern developers have gravitated toward is using MarkLogic Server to create single-tier applications, where an XQuery program accesses the content in a database, prepares it for display to an application, and sends the results to a client over an HTTP App Server.

Many of the extensions in the `1.0-ml` enhanced XQuery dialect make building these types of applications easier and more efficient. Extensions to the language such as try/catch are very useful in building robust applications. For details on these extensions, see “MarkLogic Server Enhanced XQuery Language” on page 14.

The *Developer’s Guide* lists many common design patterns in MarkLogic Server. It provides details about searches, lexicons, and many other techniques developers use to build applications in MarkLogic Server.

7.2 Using Functions

Functions are a powerful way to encapsulate XQuery code. For an example of an XQuery function, see “Declaring Functions” on page 27. This section covers the following aspects of functions:

- Creating Reusable and Modular Code
- Recursive Functions

7.2.1 Creating Reusable and Modular Code

Functions provide a convenient way to modularize or componentize your XQuery code. When you move some functionality into a function in a library module, it allows you to call that library module and use any of its functions from any other XQuery module, allowing maximum code reuse. You can separate the library modules any way that makes sense for your development environment. For example, you can use a model-view-controller (MVC) approach where you have a set of functions that are used to access the content, a set of functions used to display the content in a user-interface, and a set of functions used to control the business logic of the application (for example, workflow logic based on various events).

7.2.2 Recursive Functions

Using functions recursively (creating functions that call themselves) is a useful design pattern in XQuery. Recursive functions are very convenient for iterating through an XML tree structure to perform XML transformations from one structure to another. For details on performing recursive transformations, see the Transforming XML Structures With a Recursive typeswitch Expression chapter of the *Developer's Guide*.

7.3 Search Functions

MarkLogic Server includes functions to perform high-performance full-text search queries. The `cts:query` constructors allow you to compose complex queries. The `cts:search` API returns relevance-ranked, search-engine style queries. The `cts:contains` API can be used in XPath predicates or other XQuery expressions. Both cts:search and `cts:contains` take the composable `cts:query` APIs as a parameter, allowing you to perform full-text searches in any XQuery context, whether it is on content stored in a database or on content constructed in memory.

There are many index settings on the database configuration. The indexes speed up searches (both XPath and `cts:search`) on documents in the database. The default index settings provide a good mix of performance and economy of disk space, and the default settings work well in many applications. If you want more index options, you can configure them at the database level.

For details on composing `cts:query` constructors, see Composing cts:query Expressions in the *Developer's Guide*. For the syntax of the various search built-in functions, see *XQuery Built-In and Module API Documentation*. For details on index options, see the Databases and Text Indexing chapters of the *Administrator's Guide*.

7.4 Updates and Transactions

MarkLogic Server is a transactional system that ensures data integrity. When you perform updates on documents in a database, the system automatically locks any needed documents to ensure those documents are not updated by any other concurrent transactions. If a query reads a document, the system ensures that it the query reads a consistent view of the document throughout the transaction.

There are XQuery functions built into MarkLogic Server to create documents, update documents, and delete documents in a database. These update built-in functions are used in XQuery programs, so you can build complex logic (or whatever is required by your application) into your programs that update content.

For details on transactions, see the Understanding Transactions in MarkLogic Server chapter in the *Developer's Guide*. For details on the update built-in functions, see the *XQuery Built-In and Module API Documentation*.

7.5 HTTP App Server Functions

When you issue XQuery requests against a MarkLogic Server HTTP App Server, the requests are processed over the HTTP protocol. MarkLogic Server provides XQuery built-in functions to perform various HTTP server functions. Use these functions to HTTP-server related actions such as adding an HTTP header, accessing the request object, and so on.

The App Server functions are extremely useful when you are creating complete applications that return XHTML. For details about the signatures of the App Server functions, see *XQuery Built-In and Module API Documentation*.

7.6 Additional Resources

This section lists some sources for additional XQuery resources. They include:

- MarkLogic Server Documentation
- XQuery Use Cases
- Other Publications

7.6.1 MarkLogic Server Documentation

In addition to this document, which describes the XQuery language implemented in MarkLogic Server, the MarkLogic Server documentation also includes XQuery API documentation for all of the XQuery-standard functions as well as the Mark Logic-defined XQuery functions. Included in the API documentation are many useful XQuery code samples.

The other documents in the MarkLogic Server library describe various other aspects of the product. In particular, the *Developer's Guide* includes many useful XQuery design patterns that work well with MarkLogic Server. For a description of MarkLogic Server documentation, see the product documentation section of the Mark Logic Developer site (http://developer.marklogic.com/).

7.6.2 XQuery Use Cases

MarkLogic Server includes an application that shows the XQuery Use Cases. The Use Cases have been developed by the W3C XQuery Working Group and demonstrates how a significant number of core tasks can be implemented using the XQuery language. The W3C describes the use cases in the following document:

http://www.w3.org/TR/xquery-use-cases/

When you install MarkLogic Server, an online version of these use cases is installed on an App Server running at port 8000. If you have MarkLogic Server installed on your local machine, enter the following URL in a browser to access the use cases:

http://localhost:8000/

This application provides an initial opportunity to get your hands on the XQuery code—looking at sample XQuery, executing it against sample datasets, and even modifying it in place to see what happens when you start to change the XQuery code snippets yourself and then execute them. The Use Cases application has a default XQuery dialect of 1.0, so if you want to run code in 1.0-ml, use an XQuery version declaration in the prolog, as described in "Specifying the XQuery Dialect in the Prolog" on page 11. The *Getting Started with MarkLogic Server* walks you through this process of using the Use Cases application some detail.

7.6.3 Other Publications

In addition to the MarkLogic Server documentation, there are many excellent third-party books on XQuery. See the Mark Logic developer site for some recommendations (http://developer.marklogic.com).

You can also look directly at the XQuery specification, although much of the specification is geared more toward people who are implementing an XQuery processor rather than for people who are writing applications in XQuery. Nevertheless, it is very useful to at least get some familiarity with the following specifications:

- The current XQuery language recommendation (http://www.w3.org/TR/xquery/).
- The current recommendation for XQuery Functions and Operators (http://www.w3.org/TR/xquery-operators/).
- The XML Schema standard—useful for both type definitions and to understand the schema definitions that can be used in MarkLogic Server.

8.0 Technical Support

Mark Logic provides technical support according to the terms detailed in your Software License Agreement. For evaluation licenses, Mark Logic may provide support on an "as possible" basis.

We invite you to visit our support website at http://support.marklogic.com to access our full suite of documentation and help materials.

If you have questions or comments, you may contact Mark Logic Technical Support at the following email address:

support@marklogic.com

If reporting a query evaluation problem, please be sure to include the sample XQuery code.