XQuark Bridge 1.1

XQuery Reference Guide

XQUARK BRIDGE 1.1

XQUERY REFERENCE GUIDE

Document version 1.1

FOREWORD

Status of the W3C references from which this document derives :

XML Schema Part 1: Structures. W3C Recommendation 2 May 2001. See http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/

XML Schema Part 2: Datatypes. W3C Recommendation 2 May 2001. See http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/

Namespaces in XML. W3C Recommendation 14 January 1999. See http://www.w3.org/TR/1999/REC-xml-names-19990114

XML Information Set. W3C Recommendation 24 October 2001. See http://www.w3.org/TR/2001/REC-xml-infoset-20011024

XML Path Language (XPath) version 1.0. W3C Recommendation 16 November 1999. See http://www.w3.org/TR/1999/REC-xpath-19991116

XQuery 1.0 An XML Query Language. W3C Working Draft 12 November 2003. See http://www.w3.org/TR/2003/WD-xquery-20031112

XQuery 1.0 and XPath 2.0 Functions and Operators. W3C Working Draft 12 November 2003. See http://www.w3.org/TR/2003/WD-xpath-functions-20031112/

XQuery 1.0 and XPath 2.0 Data Model. W3C Working Draft 12 November 2003. See http://www.w3.org/TR/2003/WD-xpath-datamodel-20031112/

XML Query Use Cases. W3C Working Draft 12 November 2003. See http://www.w3.org/TR/2003/WD-xquery-use-cases-20031112/

XQuery 1.0 Formal Semantics. W3C Working Draft 12 November 2003. See http://www.w3.org/TR/2003/WD-xquery-semantics-20031112/

XSLT 2.0 and XQuery 1.0 Serialization. W3C Working Draft 12 November 2003. See http://www.w3.org/TR/2003/WD-xslt-xquery-serialization-20031112/

This section describes the status of these documents from W3C at the time of their publication. Other documents may supersede these documents. The latest status of these document series is maintained at the W3C.

Table of contents

Part

Abstract

XQuery is an XML query language designed by the W3C to be broadly applicable across a large variety of native or non-native XML data sources, including structured and semi-structured documents, relational databases, and object repositories. XQuark Bridge provides an implementation compatible with XQuery and applicable to relational data sources. It is a middleware that wraps a relational database into an XML view, which can then be queried using XQuery.

This document is the query reference guide for XQuark Bridge: it describes valid expressions of the language, as well as the specific, relational-backed XML data model to which the queries are applied.

Part

2

Introduction

Overview

XML has established itself as the standard data exchange format between applications on the Intranet and on the Internet. This has created the need for applications to publish their data in XML. As a large amount of today's business data is stored in relational databases, a general way of publishing relational data in XML is required. This requirement has been taken into consideration when designing the XQuery language: XQuery is an XML query language designed by the World-Wide Web Consortium[1] (W3C) to be broadly applicable across a large variety of native or non-native XML data sources, including structured and semi-structured documents, relational databases, and object repositories. XQuery is currently work in progress at the W3C. This user guide is based on the Working Draft dated November 12, 2003, which comprises four main documents [XQuery 1.0 An XML Query Language], [XQuery 1.0 and XPath 2.0 Functions and Operators], [XQuery 1.0 and XPath 2.0 Data Model] and [XML Query Use Cases].

XQuark Bridge provides an implementation compatible with XQuery and applicable to relational data sources. It does so by defining a generic XML view on top of a relational database schema, and by querying this view using XQuery expressions and built-in functions.

XQuark Bridge provides mechanisms to:

- Expose a subset of a relational schema as an XML database. This XML view exports its metadata information as a strongly-typed XML schema.
- Allow relational tables and views to be queried as collections of XML documents, using XQuery syntax.
- Execute strongly-types XQuery queries over the exposed XML collections.
- Return query results as newly constructed XML documents.

The above mechanisms represent a complete framework for efficiently publishing relational data in XML.

[1] The W3C is an organisation, widely supported by the industry, in charge of defining Internet-related standards, including XML and derived standards.

This reference guide is organized in four main sections:

- A description of the XML data model which is constructed by XQuark Bridge from the underlying relational model, and which defines the XML information available to the query processor.
- A description of the overall syntax used to express XQueries and their evaluation context.
- A guide to the XQuery language expressions, or more precisely to the subset of the XQuery expressions which is used by XQuark Bridge to query relational data.
- A list of available built-in functions.

Notations and conventions

This section introduces the typography used to present technical information in this manual.

The XQuark Bridge configuration files use a specific XML vocabulary to describe configuration options. In the XML representation, bold-face attribute names indicate a required attribute information item, and the rest are optional. Where an attribute information item has an enumerated type definition, the values are shown separated by vertical bars; if there is a default value, it is shown following a colon.

The allowed content of the information item is shown as a grammar fragment, using the Kleene operators `?` (0 or 1 occurrence), `*` (0 or more occurences) and + (1 or more occurrences).

```
<datasource
....name = xs:string>
....Content: (description?, url, user?, password?,
              substitutions?, catalog*)
</datasource>
```

The XML Schema that formally defines the XML vocabulary for configuration files is provided in Appendix A - XML Schema for the XQuark Bridge configuration file. This schema is associated to the namespace identified by the following URI: `http://www.xquark.org/Bridge/1.0/Datasource`.

XQuery expressions are described using grammar productions, based on a basic EBNF notation:

```
Query                ::= QueryProlog Expr
QueryProlog          ::= (NamespaceDecl
                          | DefaultNamespaceDecl)*
```

```
NamespaceDecl        ::= "namespace" NCName "="
                         StringLiteral
DefaultNamespaceDecl ::= "default element namespace ="
                         StringLiteral
```

Grammar productions within the body of the manual use only non-terminals, and all terminals are expanded for readability. Some basic non-terminals, defined in [XML Names] (e.g `QName` or `NCName`) are not defined in the manual body, but are present in the complete grammar for the XQuery language supported by XQuark Bridge, given in Appendix B – Complete BNF Grammar.

Examples are provided throughout this manual as code listings, for instance:

```
for $u in collection("USERS")/USERS,
    $i in collection("ITEMS")/ITEMS
where $u/USERID = $i/OFFERED_BY
return
  <result>
    { $u/NAME }
    { $i/DESCRIPTION }
  </result>
```

Important notes, such as standard compliance notes, are presented as:

> **Note**: The JDBC type used when constructing the XML type represents the native type of the column in the database, not necessarily the one specified in the table creation statement. For instance, Oracle replaces all ANSI column type specifications by its own native types at table creation time.

References

[XML Schema Part 1] *XML Schema Part 1: Structures*. W3C Recommendation 2 May 2001. See http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/

[XML Schema Part 2] *XML Schema Part 2: Datatypes*. W3C Recommendation 2 May 2001. See http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/

[XML Names] *Namespaces in XML*. W3C Recommendation 14 January 1999. See http://www.w3.org/TR/1999/REC-xml-names-19990114

[XML Infoset] *XML Information Set*. W3C Recommendation 24 October 2001. See http://www.w3.org/TR/2001/REC-xml-infoset-20011024

[XPath1.0] XML Path Language (XPath) version 1.0. W3C Recommendation 16 November 1999. See http://www.w3.org/TR/1999/REC-xpath-19991116

[XQuery 1.0 An XML Query Language] *XQuery 1.0 An XML Query Language*. W3C Working Draft 30 Avril 2002. See http://www.w3.org/TR/2002/WD-xquery-20020430

[XQuery 1.0 and XPath 2.0 Functions and Operators] *XQuery 1.0 and XPath 2.0 Functions and Operators*. W3C Working Draft 12 November 2003. See http://www.w3.org/TR/2003/WD-xpath-functions-20031112/

[XQuery 1.0 and XPath 2.0 Data Model] *XQuery 1.0 and XPath 2.0 Data Model*. W3C Working Draft 12 November 2003. See http://www.w3.org/TR/2003/WD-xpath-datamodel-20031112/

[XML Query Use Cases] *XML Query Use Cases*. W3C Working Draft 12 November 2003. See http://www.w3.org/TR/2003/WD-xquery-use-cases-20031112/

[XQuery 1.0 Formal Semantics] *XQuery 1.0 Formal Semantics*. W3C Working Draft 12 November 2003. See http://www.w3.org/TR/2003/WD-xquery-semantics-20031112/

[XSLT 2.0 and XQuery 1.0 Serialization] *XSLT 2.0 and XQuery 1.0 Serialization*. W3C Working Draft 12 November 2003. See http://www.w3.org/TR/2003/WD-xslt-xquery-serialization-20031112/

3

XQuery Data Model for XQuark Bridge

This section describes the XML view, defined on top of the relational database, which can be queried through XQuark Bridge.

The standard XQuery data model

XQuery introduces an XML data model, which defines precisely the information in an XML document that is available to an XQuery processor. It also defines all permissible values of expressions in the XQuery language.

The XQuery data model extends existing XML data models, such as the ones defined in XML Information Set [XML Infoset] or in XPath 1.0 [XPath1.0] by adding two new features to the model:

- Support for XML Schema types: XML elements, attributes and text nodes can be associated to structured complex types and simple data types, as defined in the XML Schema Recommendation (resp. [XML Schema Part 1] and [XML Schema Part 2]).
- Representation of collections of documents and complex elements.

Every value handled by the data model is a *sequence* of zero or more *items*. An item is either a *node* or an *atomic value*.

A *node* is one of seven node kinds, as in the XPath 1.0 data model: document, element, attribute, namespace, processing-instruction, comment, text.

An *atomic value* encapsulates an XML Schema simple type and a corresponding value of that type.

A *sequence* is an ordered collection of nodes, atomic values, or any mixture of nodes and atomic values. A sequence cannot be a member of a sequence. A single item appearing on its own is modeled as a sequence containing one item.

The XQuery data model can represent various values including not only the input and the output of a query, but all values of expressions used during the intermediate calculations. Examples include the input document or

document collection (represented as a document node or a sequence of document nodes), the result of a path expression (represented as a sequence of nodes), the result of an arithmetic or a logical expression (represented as an atomic value), a sequence expression resulting in a sequence of integers, dates, QNames or other XML Schema atomic values (represented as a sequence of atomic values), etc.

A complete specification of the XQuery data model can be obtained in the W3C draft recommendation [XQuery 1.0 and XPath 2.0 Data Model].

Typing in the standard XQuery data model

The XQuery data model relies on the XML Information Set obtained after XML Schema validity assessment. XML Schema validity assessment is the process of assessing an XML element information item with respect to an XML Schema and augmenting it and some or all of its descendants with properties that provide information about validity and type assignment. The result of schema validity assessment is an augmented Infoset, known as the Post Schema-Validation Infoset, or PSVI. The type information associated to each element node, attribute node or atomic value involves *schema components* of four different kinds: *element declaration*, *attribute declaration*, *complex type* and *simple type*, as defined in [XML Schema Part 1].

If validity has been successfully assessed, the item is guaranteed to be a valid instance of its associated type as defined by XML Schema. If not (either because no schema information was available or because the item is invalid), the type of the item is unknown.

Every node has a **string value**, which is the textual content of the node, as in XPath 1.0, and a **typed value**, which is a sequence of atomic values. The typed value for the various kinds of nodes is defined as follows:

- The typed value of a document, namespace, comment, text or processing instruction node is its string value.
- The typed value of an element or attribute node that has no type annotation is its string value.
- The typed value of an element or attribute node whose type annotation denotes either a simple type or a complex type with simple content is a sequence of atomic values that is obtained by transforming the string content of the node into the value space of the associated type, as specified in [XML Schema Part 2].
- The typed value of an element node whose type annotation denotes a complex type with complex content is not defined, accessing it raises an error.

The XQuark Bridge data model

The default XQuark Bridge data model is obtained by mapping relational structures and data into the XQuery data model introduced above. The result is an XML view where :

- The relational structures are exposed as one or several XML schemas, which contain all the *schema components* that are derived from the relational model. Those components are *element declarations*, which describe the XML structure of the content of each published relational table. By default, when accessing a single relational container[2], the generated schema does not have a target namespace. When accessing several containers, one XML schema is generated per container, and must necessarily be associated with a user-specified target namespace.
- Each exposed relational table is viewed as a named *collection of documents*, whose name is by default the name of the underlying relational object, optionally prefixed with the name of the relational container to which it belongs. Prefix and name are separated by a dot (e.g. ORDERS.CUSTOMER), as usual in relational databases.
- Each row in the relational table is viewed as a *document node*. The document has a top-level element which is schema-valid with respect to the element declaration generated from the table structure.

XQuark Bridge not only supports relational tables, but also views and synonyms, in a similar way. However, there are some limitations to the use of views and synonyms, as rows in those structures cannot be easily associated to identifiers. Those limitations appear when views or synonyms are used in nested queries, and are further detailed in the FLWR Expressions section of this manual.

XQuark Bridge also provides a way to control the XML view generation, by providing support for filtering and renaming tables and columns. This control is specified through a configuration file, whose syntax is detailed in Controlling the XML view generation.

Generated schema components

Each published table is associated to a generated element declaration, which provides an XML view of the table relational structure. The rules for generating the element declaration are listed below:

- The element declaration *name* is by default the name of the table, as internally represented in the database metadata. Unlike some relational databases, XML is case-sensitive, so a table name represented in upper case in the database will have to be used exclusively in upper case in

[2] A relational container is usually called a schema or a catalog, depending on the relational database vendor.

queries. Table names that are not legal XML element names (e.g. those containing '$' or '#' signs) are ignored by XQuark Bridge, unless they are explicitly renamed in the XQuark Bridge configuration file, as detailed below. If the container of the table is associated to a target namespace, this namespace is the element declaration namespace, otherwise the element declaration does not have a namespace.

- The element declaration *type* is a complex type, whose content is a sequence containing a nested element declaration for each published column in the table.
- Each nested element declaration *name* is by default the name of the column, as internally represented in the database metadata. Column names that are not legal XML element names (e.g. those containing '$' or '#' signs) are ignored by XQuark Bridge, unless they are explicitly renamed in the XQuark Bridge configuration file, as detailed below. Nested element declarations are always considered local to their enclosing element, and therefore do not have a namespace.
- Each nested element declaration *type* is a predefined XML Schema simple type, obtained from the column JDBC type according to the table below. When a JDBC type is unsupported, the column is ignored (i.e. no nested element declaration will appear in the complex type).

JDBC Type	**XML type**
ARRAY	*not supported*
BIGINT	xs:long
BINARY	xs:base64Binary (with *length* attribute)
BIT	xs:boolean
BLOB	xs: base64Binary (with *maxLength* attribute)
CHAR	xs:string (with *length* attribute)
CLOB	xs:string (with *maxLength* attribute)
DATE	xs:date
DECIMAL	xs:decimal (with *totalDigits* and *fractionDigits* attributes)
DISTINCT	*not supported*
DOUBLE	xs:double
FLOAT	xs:double
INTEGER	xs:integer
JAVA_OBJECT	*not supported*
LONGVARBINARY	xs:base64Binary (with *maxLength* attribute)

LONGVARCHAR	xs:string (with *maxLength* attribute)
NULL	*not supported*
NUMERIC	xs:decimal (with *totalDigits* and *fractionDigits* attributes)
OTHER	*not supported*
REAL	xs:float
REF	*not supported*
SMALLINT	xs:short
STRUCT	*not supported*
TIME	xs:time
TIMESTAMP	xs:dateTime
TINYINT	xs:byte
VARBINARY	xs:base64Binary (with *maxLength* attribute)
VARCHAR	xs:string (with *maxLength* attribute)

> **Note**: The JDBC type used when constructing the XML type represents the native type of the column in the database, not necessarily the one specified in the table creation statement. For instance, Oracle replaces all ANSI column type specifications by its own native types at table creation time.

Example

As an example, consider a relational database used by an online auction. The auction maintains a USERS table containing information on registered users, each identified by a unique userid, who can either offer items for sale or bid on items. An ITEMS table lists items currently or recently for sale, with the userid of the user who offered each item. A BIDS table contains all bids on record, keyed by the userid of the bidder and the item number of the item to which the bid applies.

The relational model for this example is defined below:

```
CREATE TABLE USERS (
  USERID        CHAR(3) PRIMARY KEY,
  NAME          VARCHAR(20) UNIQUE,
  RATING        CHAR(1)
);

CREATE TABLE ITEMS (
  ITEMNO        CHAR(4) PRIMARY KEY,
  DESCRIPTION   VARCHAR(30),
  OFFERED_BY    CHAR(3) REFERENCES USERS(USERID),
  START_DATE    DATE,
```

```
  END_DATE      DATE,
  RESERVE_PRICE NUMBER(10)
);

CREATE TABLE BIDS (
  USERID        CHAR(3) REFERENCES USERS(USERID),
  ITEMNO        CHAR(4) REFERENCES ITEMS(ITEMNO),
  BID           NUMBER(10) NOT NULL,
  BID_DATE      DATE
);
```

The data for this example is given in the three tables below:

USERS		
USERID	NAME	RATING
U01	Tom Jones	B
U02	Mary Doe	A
U03	Dee Linquent	D
U04	Roger Smith	C
U05	Jack Sprat	B
U06	Rip Van Winkle	B

ITEMS					
ITEMNO	DESCRIPTION	OFFERED_BY	START_DATE	END_DATE	RESERVE_PRICE
1001	Red Bicycle	U01	99-01-05	99-01-20	40
1002	Motorcycle	U02	99-02-11	99-03-15	500
1003	Old Bicycle	U02	99-01-10	99-02-20	25
1004	Tricycle	U01	99-02-25	99-03-08	15
1005	Tennis Racket	U03	99-03-19	99-04-30	20
1006	Helicopter	U03	99-05-05	99-05-25	50000
1007	Racing Bicycle	U04	99-01-20	99-02-20	200
1008	Broken Bicycle	U01	99-02-05	99-03-06	25

BIDS			
USERID	ITEMNO	BID	BID_DATE
U02	1001	35	99-01-07
U04	1001	40	99-01-08
U02	1001	45	99-01-11
U04	1001	50	99-01-13
U02	1001	55	99-01-15
U01	1002	400	99-02-14
U02	1002	600	99-02-16
U03	1002	800	99-02-17
U04	1002	1000	99-02-25

BIDS			
USERID	ITEMNO	BID	BID_DATE
U02	1002	1200	99-03-02
U04	1003	15	99-01-22
U05	1003	20	99-02-03
U01	1004	40	99-03-05
U03	1007	175	99-01-25
U05	1007	200	99-02-08
U04	1007	225	99-02-12

The XML schema generated by XQuark Bridge[3] for this example is shown below:

```
<?xml version='1.0'?>

<schema xmlns="http://www.w3.org/2001/XMLSchema">

      <element name="BIDS">
        <complexType>
          <sequence>
            <element name="USERID" minOccurs="0">
              <simpleType>
                <restriction base="string">
                  <length value="3"/>
                </restriction>
              </simpleType>
            </element>
            <element name="ITEMNO" minOccurs="0">
              <simpleType>
                <restriction base="string">
                  <length value="4"/>
                </restriction>
              </simpleType>
            </element>
            <element name="BID">
              <simpleType>
                <restriction base="decimal">
                  <totalDigits value="10"/>
                  <fractionDigits value="0"/>
                </restriction>
              </simpleType>
            </element>
            <element name="BID_DATE"
                      minOccurs="0"
                      type="dateTime"/>
          </sequence>
        </complexType>
      </element>

  <element name="ITEMS">
    <complexType>
```

[3] This schema corresponds to the metadata information returned by the Oracle database for the relational model shown above. Other databases might create slightly different XML schemas.

```
      <sequence>
        <element name="ITEMNO">
          <simpleType>
            <restriction base="string">
              <length value="4"/>
            </restriction>
          </simpleType>
        </element>
        <element name="DESCRIPTION"
                 minOccurs="0">
          <simpleType>
            <restriction base="string">
              <maxLength value="30"/>
            </restriction>
          </simpleType>
        </element>
        <element name="OFFERED_BY" minOccurs="0">
          <simpleType>
            <restriction base="string">
              <length value="3"/>
            </restriction>
          </simpleType>
        </element>
        <element name="START_DATE" minOccurs="0"
                 type="dateTime"/>
        <element name="END_DATE" minOccurs="0"
                 type="dateTime"/>
        <element name="RESERVE_PRICE"
                 minOccurs="0">
          <simpleType>
            <restriction base="decimal">
              <totalDigits value="10"/>
              <fractionDigits value="0"/>
            </restriction>
          </simpleType>
        </element>
      </sequence>
    </complexType>
  </element>

  <element name="USERS">
    <complexType>
      <sequence>
        <element name="USERID">
          <simpleType>
            <restriction base="string">
              <length value="3"/>
            </restriction>
          </simpleType>
        </element>
        <element name="NAME" minOccurs="0">
          <simpleType>
            <restriction base="string">
               <maxLength value="20"/>
            </restriction>
          </simpleType>
        </element>
        <element name="RATING" minOccurs="0">
```

```
          <simpleType>
            <restriction base="string">
              <length value="1"/>
            </restriction>
          </simpleType>
        </element>
      </sequence>
    </complexType>
  </element>

</schema>
```

Controlling the XML view generation

As described above, XQuark Bridge publishes a relational schema as a generic, strongly typed XML view, which can then be used as the basis for running XQueries. Although this generic approach is convenient in many situations, there are cases where finer control on the XML view generation is required. Those cases include:

- Applications which access several relational schemas,
- Applications which access only a small fraction of the relational tables in a relational schema,
- Applications which access tables and columns whose names are not legal XML element names.

For the benefit of those applications, XQuark Bridge provides configuration files that allow the application designer to better control the generated XML view. Configuration files are written in XML. Each file describes the wrapping of a single relational datasource, defined by the JDBC triple { JDBC URL, user, password }.

Selecting and filtering relational structures

The general structure of a configuration file is shown below:

```
<datasource name="{ Datasource identifier }">
  <description>
    { Optional datasource description }
  </description>
  <url> { JDBC connection string } </url>
  <user> { User name } </user>
  <password> { User password } </password>

  <substitutions>
    <nameCase>
      { lower | upper | mixed }
    </nameCase>
    <character value="{ character }"
               subst="{ substitution string }"/>
    …
  </substitutions>
```

```
  <catalog name="{ Optional catalog name }">
    <schema name="{ Optional schema name }"
            targetNamespace="{ Namespace URI }"
            elementFormDefault="{ qualified
                                | unqualified }">
      <includes>
        <table regex="{ Regular expression }"/>
        <table name="{ Table name }"
               alias="{ Table alias }">
          <includes>
            <column
              regex="{ Regular expression }"/>
            <column name="{ Column name }
                    alias="{ Column alias }"/>
          </includes>
          <excludes>
            <column
              regex="{ Regular expression }"/>
            <column name="{ Column name }"/>
          </excludes>
        </table>
      </includes>
      <excludes>
        <table regex="{ Regular expression }"/>
        <table name="{ Table name }" />
      </excludes>
    </schema>
  </catalog>
</datasource>
```

The complete XML Schema for the configuration file is given in Appendix A - XML Schema for the XQuark Bridge configuration file.

Three main sections appear in the configuration file:

- The datasource declaration,
- The substitutions declaration,
- The selection of the catalogs, schemas, tables and columns to be used in the XML view.

The datasource declaration section is composed of the following elements:

```
<datasource
....name = xs:string>
....Content: (description?, url, user?, password?,
              substitutions?, catalog*)
</datasource>

<description>
    Content: xs:string
</description>
```

```
<url>
    Content: xs:string
</url>

<user>
    Content: xs:string
</user>
```

```
<password>
    Content: xs:string
</password>
```

The role of the above elements and attributes is detailed below:

- The `name` attribute is an identifier defined by the application designer for this particular datasource.
- The optional `description` element is present for documentation purpose.
- The mandatory `url` element identifies the database instance to be wrapped.
- The optional `user` and `password` elements are used for the connection to the wrapped database instance.

The substitutions section is detailed in the next section.

The selection section is a set of hierarchical elements that represent traditional relational concepts:

```
<catalog
....name = xs:string>
....Content: schema+
</catalog>

<schema
....name = xs:string
    targetNamespace = xs:anyURI
    elementFormDefault = qualified | unqualified
                       : unqualified>
    Content: (includes?, excludes?)
</schema>

<includes>
    Content: table+ | column+
</includes>

<excludes>
    Content: table+ | column+
</excludes>

<table
    name = xs:string
    regex = xs:string
```

```
    alias = xs:NCName>
    Content: (includes?, excludes?)
</table>
```

```
<column
    name = xs:string
    regex = xs:string
    alias = xs:NCName />
```

The role of the above elements and attributes is detailed below:

- The `catalog` element can appear one or several times in the `datasource` top-level element, and represents a relational catalog in the wrapped database instance. The `name` attribute is optional and must be omitted if the database does not support the catalog concept: in this latter case, only a single, anonymous `catalog` element should appear in the configuration file. On the other hand, if more than one catalog are to be selected, each `catalog` element should have a `name` attribute, which represents the name of the catalog to be selected.
- The `schema` element can appear one or several times in a `catalog` element, and represents a relational schema in the enclosing catalog. The `name` attribute is optional and must be omitted if the database does not support the schema concept: in this latter case, only a single, anonymous `schema` element should appear in each `catalog` element. On the other hand, if more than one schema are to be selected, each `schema` element should have a `name` attribute, which represents the name of the schema to be selected. In addition, each `schema` element can have a `targetNamespace` attribute, which specifies the namespace of the generated element declarations associated to the schema tables. This attribute is optional only when XQuark Bridge accesses a single relational schema. In all other cases, each individual schema must be associated to a target namespace. When a target namespace is specified, an additional optional `elementFormDefault` attribute can be used to control the qualification of the inner generated element declarations (the ones corresponding to the table columns): if the attribute value is `qualified`, inner element declarations are qualified with the target namespace; if the attribute value is `unqualified`, or the attribute is absent, inner element declarations are not qualified.
- The `includes` element can appear zero or one time in a `schema` element (resp. a `table` element). It is used as a container for the elements that select tables (resp. columns) to be included in the generated XML view. When the `includes` element is absent, all tables (resp. columns) contained in the enclosing object are included.
- The `excludes` element can appear zero or one time in a `schema` element (resp. a `table` element). It is used as a container for the elements that select tables (resp. columns) to be excluded in the generated XML view.. Exclusion has higher priority than inclusion: a

table which is both included and excluded will not appear in the XML view.

- The `table` element can appear one or several times in an `includes` or `excludes` element, and is used to select tables, views or synonyms[4] in the enclosing schema. One and only one of the `name` or `regex` attribute must be present. The `name` attribute selects the table of the given name in the enclosing schema: if no table corresponding to the name is found, an error is generated. The `regex` attribute selects all the tables in the enclosing schema that have a name matching the specified regular expression. This regular expression uses the grammar described in [XML Schema Part 2], which is very close to the regular expression syntax of the Perl language. No error is generated if no match is found for the regular expression. When the `name` attribute is used, the element declaration generated for the table can be further refined by using an `includes` and/or `excludes` nested element to specify the columns to be used. This possibility is not available when the `regex` attribute is used. The use of the `alias` attribute is described in next section.

- The `column` element can appear one or several times in an `includes` or `excludes` element, and is used to select columns in the enclosing table. One and only one of the `name` or `regex` attribute must be present. The `name` attribute selects the column of the given name in the enclosing table: if no column corresponding to the name is found, an error is generated. The `regex` attribute selects all the columns in the enclosing table that have a name matching the specified regular expression. No error is generated if no match is found for the regular expression. The use of the `alias` attribute is described in next section.

Using aliases to rename relational structures

The XQuark Bridge configuration file also provides support for renaming relational structures. Renaming can be useful when:

- Table or column names contain characters that are not legal XML element names.
- Generated element declarations must match a predefined XML schema.

While both capabilities can be obtained using traditional relational database techniques such as views or synonyms, XQuark Bridge offers an additional level of flexibility through the configuration file.

Aliases for table and column names can be specified in two ways:

- Globally, by associating a substitution string to each unsupported character. XQuark Bridge will automatically substitute the string each time the character is encountered in a table or column name. The case of the generated table or column names can also be controlled globally.

[4] In the following discussion, table is used to represent a relational table, view or catalog.

- Locally, by associating an alias to a specific table or column.

Global substitutions are specified by adding an optional `substitutions` element after the datasource declaration section in the configuration file.

```
<substitutions>
    Content: (nameCase?, character*)
</substitutions>

<nameCase>
    Content: text() = mixed | upper | lower : mixed
</nameCase>

<character
    value = xs:string
    subst = xs:string />
```

This element contains:

- an optional `nameCase` element, which specifies the case management policy: *mixed* (the default) to preserve the case of the names returned by the database, *upper* or *lower* to impose a particular policy.
- one or several `character` elements, which specify the character to be replaced (the `value` attribute) and the substitution string (the `subst` attribute).

Local substitutions are specified by adding an optional `alias` attribute to the `table` or `column` element selecting the table or column to be renamed. The value of the attribute is the alias to be used by XQuark Bridge for the relational structure: the generated element declaration associated to the table or column will have the specified alias as name. The `alias` attribute can only be used in conjunction with the `name` attribute in `table` or `column` elements: structures selected through regular expressions cannot be renamed. Alias values are case-sensitive.

Part 4

XQuery queries

Language specification reference

The XQuery Language specification implemented by XQuark Bridge is available at http://www.w3.org/TR/2003/WD-xquery-20031112/.

XQuark Bridge only implements a subset of the above draft specification. Rather than repeating the specification contents, this chapter describes the main characteristics of the XQuery Language that are **not** supported by the product.

Grammar rules are referenced by their number as defined in the specification of the XQuery Language. Unsupported grammar elements are shown in bold. Partially supported grammar elements (i.e. those supported with some restrictions) are shown in italics.

All grammar rules can be found in Appendix B.

Unsupported or partially supported features

Prolog declarations

Variable declarations

External variable declarations are not supported.

```
[38] VarDecl ::= <"declare" "variable" "$"> VarName
                 TypeDeclaration? (("{" Expr "}") |
                 "external")
```

Function declarations

External function declarations are not supported.

```
[120] FunctionDecl     <"declare" "function"> <QName "(">
                       ParamList? (")" | (<")" "as">
                       SequenceType)) (EnclosedExpr |
                       "external")
```

Functions are always inlined within the expression in which they are called: they are in fact considered as **parameterized views** by the XQuery parser. This approach induces a few restrictions on the supported expressions within a function body:

- Recursive functions are not supported, even in the case when the recursion is indirect, i.e. when there is a cyclic function call graph.
- Functions should not return expressions that cannot be handled in the `where` clause of a FLWOR. For instance, conditional expressions or computed element constructors should not be used in function return values.

Path expressions

Filter steps

Filter steps are only supported as the first step in a path expression. Furthermore, predicates are not allowed in a filter step.

```
[71] StepExpr   ::= AxisStep | FilterStep
[73] FilterStep ::= PrimaryExpr Predicates
```

Forward axes

The `following` and `following-sibling` forward axes are not supported.

```
[89] ForwardAxis ::= <"child" "::">
                   | <"descendant" "::">
                   | <"attribute" "::">
                   | <"self" "::">
                   | <"descendant-or-self" "::">
                   | <"following-sibling" "::">
                   | <"following" "::">
```

Reverse axes

The only supported reverse axis is `parent`.

```
[90] ReverseAxis ::= <"parent" "::">
                   | <"ancestor" "::">
                   | <"preceding-sibling" "::">
                   | <"preceding" "::">
                   | <"ancestor-or-self" "::">
```

Predicates

The XPath concepts of context position (position of the context item, i.e. the current node, in the current nodeset) and context size (size of the current nodeset) are not supported. Thus, predicate expressions returning numeric values are not supported either.

```
[77] Predicates     ("[" Expr "]")*
```

Kind tests

Only the `text()` and `node()` kind tests are supported in step expressions. Note however that all kind tests can by used in sequence type expressions.

```
[128] KindTest     DocumentTest
                   | ElementTest
                   | AttributeTest
                   | PITest
                   | CommentTest
                   | TextTest
                   | AnyKindTest
```

Sequence expressions

Sequence constructors

Sequence construction can only be used for literals, within the `where` clause of FLWOR expressions.

```
[40] Expr ::= ExprSingle ("," ExprSingle)*
```

Range expressions

Construction of a sequence of values using a range expression is not supported.

```
[62] RangeExpr     AdditiveExpr ( "to" AdditiveExpr )?
```

Set operations

Union, intersection and difference of sequence expressions are not supported.

```
[66] UnionExpr           ::= IntersectExceptExpr ( ("union"
                             | "|") IntersectExceptExpr )*
[67] IntersectExceptExpr    ValueExpr ( ("intersect" |
                            "except") ValueExpr )*
```

Arithmetic expressions

Integral division

The `idiv` operator is not supported.

```
[64] MultiplicativeExpr ::= UnaryExpr ( ("*" | "div" |
                            "idiv" | "mod") UnaryExpr )*
```

Comparison expressions

Node comparison

Node comparison is not supported.

```
[61] ComparisonExpr     RangeExpr ( (ValueComp
                        | GeneralComp
                        | NodeComp) RangeExpr )?
[84] NodeComp       ::= "is" | "<<" | ">>"
```

Constructors

Computed constructors

Computed constructors are only supported in the return clause of a FLWOR expression of the main module.

```
[81] ComputedConstructor ::= CompElemConstructor
                           | CompAttrConstructor
                           | CompDocConstructor
                           | CompTextConstructor
                           | CompXmlPI
                           | CompXmlComment
                           | CompNSConstructor
```

FLWOR expressions

Positional variable

Positional variables are not available.

```
[43] ForClause      ::= <"for" "$"> VarName TypeDeclaration?
                        PositionalVar? "in" ExprSingle ("," 
                        "$" VarName TypeDeclaration?
                        PositionalVar? "in" ExprSingle)*
[44] PositionalVar      "at" "$" VarName
```

Stable order

Stable order is not supported.

```
[47] OrderByClause     (<"order" "by"> | <"stable" "order"
                       "by">) OrderSpecList
```

Order modifier

Only the `ascending` and `descending` modifiers are supported.

```
[50] OrderModifier     ("ascending" | "descending")?
                       (<"empty" "greatest"> | <"empty"
                       "least">)? ("collation"
                       StringLiteral)?
```

Conditional expressions

Conditional (If-Then-Else) expressions can only be used in the return clause of a FLWOR expression of the main module. The condition expression should involve only simple path expressions starting with a variable, literals and built-in functions using the same type of expressions as arguments. Expressions used in the `then` and `else` clauses should not contain nested sub-queries.

```
[54] IfExpr ::= <"if" "("> Expr ")" "then" ExprSingle "else"
                ExprSingle
```

Expressions on Sequence Types

Typeswitch

Type switch expressions are not supported.

```
[41] ExprSingle      ::= FLWORExpr
                         | QuantifiedExpr
                         | TypeswitchExpr
                         | IfExpr
                         | OrExpr
[52] TypeswitchExpr      <"typeswitch" "("> Expr ")"
                         CaseClause+ "default" ("$" VarName)?
                         "return" ExprSingle
[53] CaseClause      ::= "case" ("$" VarName "as")?
                         SequenceType "return" ExprSingle
```

Instance of

The expression type cannot be tested.

```
[57] InstanceofExpr     TreatExpr ( <"instance" "of">
                        SequenceType )?
```

Casting operations

Static and dynamic casting is not supported.

```
[58] TreatExpr    ::= CastableExpr ( <"treat" "as">
                      SequenceType )?
[59] CastableExpr ::= CastExpr ( <"castable" "as">
                      SingleType )?
[60] CastExpr     ::= ComparisonExpr ( <"cast" "as">
                      SingleType )?
```

Validate expressions

Explicit validation expressions are not supported. Note however that implicit validation, based on the global validation mode, in-scope schema definitions and contructed element QNames, is fully supported.

```
[68] ValueExpr     ::= ValidateExpr | PathExpr
[78] ValidateExpr  ::= (<"validate" "{"> | (<"validate"
                       "global"> "{") | (<"validate"
                       "context"> SchemaContextLoc "{") |
                       (<"validate" SchemaMode>
                       SchemaContext? "{")) Expr "}"
[79] SchemaContext     ("context" SchemaContextLoc) |
                       "global"
```

Part

5

Supported built-in functions

Supported build-in functions

The following functions are supported by XQuark Bridge:

- fn:abs
- fn:avg
- fn:ceiling
- fn:collection
- fn:concat
- fn:contains
- fn:count
- fn:current-date
- fn:current-datetime
- fn:current-time
- fn:data
- fn:deep-equals
- fn:distinct-values
- fn:empty
- fn:ends-with
- fn:exists
- fn:false
- fn:floor
- fn:matches
- fn:max
- fn:min
- fn:not

- fn:number
- fn:round
- fn:starts-with
- fn:string-length
- fn:substring
- fn:sum
- fn:true
- fn:upper-case

Supported built-in type constructors

The following type constructors are supported by XQuark Bridge:

- xs:date
- xs:datetime
- xs:decimal
- xs:double
- xs:float
- xs:integer
- xs:string
- xs:time

Note that those constructors can only take string literals as parameters.

Part 6

Index

9

Appendix A – XML Schema for the XQuark Bridge configuration file

```
<?xml version="1.0"?>
<schema
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:ds="http://www.xquark.org/Bridge/1.0/Datasource"
  targetNamespace=
    " http://www.xquark.org/Bridge/1.0/Datasource ">

  <simpleType name="caseType">
    <restriction base="string">
      <enumeration value="mixed"/>
      <enumeration value="lower"/>
      <enumeration value="upper"/>
    </restriction>
  </simpleType>

  <simpleType name="elementFormType">
    <restriction base="string">
      <enumeration value="qualified"/>
      <enumeration value="unqualified"/>
    </restriction>
  </simpleType>

  <simpleType name="charType">
    <restriction base="string">
      <length value="1"/>
    </restriction>
  </simpleType>

  <simpleType name="substType">
    <restriction base="string">
      <pattern value="[\c-[:]]*" />
    </restriction>
  </simpleType>

  <complexType name="relationalStructType">
    <attribute name="regex" type="string" />
    <attribute name="name" type="string" />
  </complexType>

  <complexType name="aliasedRelationalStructType">
    <complexContent>
      <extension base="ds:relationalStructType">
        <attribute name="alias" type="NCName" />
```

```
      </extension>
    </complexContent>
  </complexType>

  <complexType name="excludedTableType">
    <complexContent>
      <extension base="ds:relationalStructType"/>
    </complexContent>
  </complexType>

  <complexType name="excludedColumnType">
    <complexContent>
      <extension base="ds:relationalStructType"/>
    </complexContent>
  </complexType>

  <complexType name="includedColumnType">
    <complexContent>
      <extension base="ds:aliasedRelationalStructType"/>
    </complexContent>
  </complexType>

  <complexType name="includedTableType">
    <complexContent>
      <extension base="ds:aliasedRelationalStructType">
        <sequence>
          <element name="includes" minOccurs="0">
            <complexType>
              <sequence>
                <element name="column" maxOccurs="unbounded"
                         type="ds:includedColumnType" />
              </sequence>
            </complexType>
          </element>
          <element name="excludes" minOccurs="0">
            <complexType>
              <sequence>
                <element name="column" maxOccurs="unbounded"
                         type="ds:excludedColumnType" />
              </sequence>
            </complexType>
          </element>
        </sequence>
      </extension>
    </complexContent>
  </complexType>

  <element name="datasource">
    <complexType>
      <sequence>
        <element name="description" minOccurs="0" type="string" />
        <element name="url" type="string" />
        <element name="user" type="string" />
        <element name="password" type="string"/>
        <element name="substitutions" minOccurs="0">
          <complexType>
            <sequence>
              <element name="nameCase" minOccurs="0"
```

```
                    type="ds:caseType" />
            <element name="character" minOccurs="0"
                     maxOccurs="unbounded">
              <complexType>
                <attribute name="value" type="ds:charType" />
                <attribute name="subst" type="ds:substType" />
              </complexType>
            </element>
          </sequence>
        </complexType>
      </element>
      <element name="catalog" maxOccurs="unbounded">
        <complexType>
          <sequence>
            <element name="schema" maxOccurs="unbounded">
              <complexType>
                <sequence>
                  <element name="includes" minOccurs="0">
                    <complexType>
                      <sequence>
                        <element name="table"
                                 maxOccurs="unbounded"
                                 type="ds:includedTableType" />
                      </sequence>
                    </complexType>
                  </element>
                  <element name="excludes" minOccurs="0">
                    <complexType>
                      <sequence>
                        <element name="table"
                                 maxOccurs="unbounded"
                                 type="ds:excludedTableType" />
                      </sequence>
                    </complexType>
                  </element>
                </sequence>
                <attribute name="name" type="string"
                           use="optional" />
                <attribute name="targetNamespace" type="anyURI"
                           use="optional" />
                <attribute name="elementFormDefault"
                           type="ds:elementFormType"
                           use="optional" />
              </complexType>
            </element>
          </sequence>
          <attribute name="name" type="string" use="optional" />
        </complexType>
      </element>
    </sequence>
    <attribute name="name" type="string"/>
  </complexType>
 </element>
</schema>
```

Appendix B – Complete BNF Grammar

Note 1 : Rule numbers are those used in XQuery language specifications.
Note 2 : Bold underlined items are not supoorted in the current version.

Named Terminals

```
[1]  Pragma               ::= "(::" "pragma" QName PragmaContents* "::)"
[2]  MUExtension          ::= "(::" "extension" QName ExtensionContents*
                              "::)"
[3]  ExprComment          ::= "(:" (ExprCommentContent | ExprComment)* ":)"
[4]  ExprCommentContent   ::= Char
[5]  PragmaContents       ::= Char
[6]  ExtensionContents    ::= Char
[7]  IntegerLiteral       ::= Digits
[8]  DecimalLiteral       ::= ("." Digits) | (Digits "." [0-9]*)
[9]  DoubleLiteral        ::= (("." Digits) | (Digits ("." [0-9]*)?)) ("e" |
                              "E") ("+" | "-")? Digits
[10] StringLiteral        ::= ('"' (PredefinedEntityRef | CharRef | ('"'
                              '"') | [^"&])* '"') | ("'"
                              (PredefinedEntityRef | CharRef | ("'" "'") |
                              [^'&])* "'")
[11] S                    ::= [http://www.w3.org/TR/REC-xml#NT-S] XML
[12] SchemaMode           ::= "lax" | "strict" | "skip"
[13] SchemaGlobalTypeName ::= "type" "(" QName ")"
[14] SchemaGlobalContext  ::= QName | SchemaGlobalTypeName
[15] SchemaContextStep    ::= QName
[16] Digits               ::= [0-9]+
[17] EscapeQuot           ::= '"' '"'
[18] PITarget             ::= NCName
[19] NCName               ::= [http://www.w3.org/TR/REC-xml-names/#NT-
                              NCName] Names
[20] VarName              ::= QName
[21] Qname                ::= [http://www.w3.org/TR/REC-xml-names/#NT-QName]
                              Names
[22] PredefinedEntityRef  ::= "&" ("lt" | "gt" | "amp" | "quot" | "apos")
                              ";"
[23] HexDigits            ::= ([0-9] | [a-f] | [A-F])+
[24] CharRef              ::= "&#" (Digits | ("x" HexDigits)) ";"
[25] EscapeApos           ::= "'"
[26] Char                 ::= [http://www.w3.org/TR/REC-xml#NT-Char] XML
[27] ElementContentChar   ::= Char - [{}<&]
[28] QuotAttContentChar   ::= Char - ["{}<&]
[29] AposAttContentChar   ::= Char - ['{}<&]
```

Non-Terminals

```
[30]  Module              ::= VersionDecl? (MainModule | LibraryModule)
[31]  MainModule          ::= Prolog QueryBody
[32]  LibraryModule       ::= ModuleDecl Prolog
[33]  ModuleDecl          ::= <"module" "namespace"> NCName "="
                              StringLiteral Separator
[34]  Prolog              ::= ((NamespaceDecl
                              | XMLSpaceDecl
                              | DefaultNamespaceDecl
                              | DefaultCollationDecl
                              | BaseURIDecl
                              | SchemaImport
                              | ModuleImport
                              | VarDecl
                              | ValidationDecl
                              | FunctionDecl) Separator)*
[35]  Separator           ::= ";"
[36]  VersionDecl         ::= <"xquery" "version" StringLiteral> Separator
[37]  ModuleImport        ::= <"import" "module"> ("namespace" NCName
                              "=")? StringLiteral <"at" StringLiteral>?
[38]  VarDecl             ::= <"declare" "variable" "$"> VarName
                              TypeDeclaration? (("{" Expr "}") |
                              "external")
[39]  QueryBody           ::= Expr
[40]  Expr                ::= ExprSingle ("," ExprSingle)*
[41]  ExprSingle          ::= FLWORExpr
                              | QuantifiedExpr
                              | TypeswitchExpr
                              | IfExpr
                              | OrExpr
[42]  FLWORExpr           ::= (ForClause | LetClause)+ HintClause?
                              WhereClause? OrderByClause? "return"
                              ExprSingle
[43]  ForClause           ::= <"for" "$"> VarName TypeDeclaration?
                              PositionalVar? "in" ExprSingle ("," "$"
                              VarName TypeDeclaration? PositionalVar? "in"
                              ExprSingle)*
[44]  PositionalVar       ::= "at" "$" VarName
[45]  LetClause           ::= <"let" "$"> VarName TypeDeclaration? ":="
                              ExprSingle ("," "$" VarName TypeDeclaration?
                              ":=" ExprSingle)*
[46]  WhereClause         ::= "where" Expr
[47]  OrderByClause       ::= (<"order" "by"> | <"stable" "order" "by">)
                              OrderSpecList
[48]  OrderSpecList       ::= OrderSpec ("," OrderSpec)*
[49]  OrderSpec           ::= ExprSingle OrderModifier
[50]  OrderModifier       ::= ("ascending" | "descending")? (<"empty"
                              "greatest"> | <"empty" "least">)?
                              ("collation" StringLiteral)?
```

```
[51]  QuantifiedExpr        ::= (<"some" "$"> | <"every" "$">) VarName
                                TypeDeclaration? "in" ExprSingle ("," "$"
                                VarName TypeDeclaration? "in" ExprSingle)*
                                "satisfies" ExprSingle
[52]  TypeswitchExpr        ::= <"typeswitch" "("> Expr ")" CaseClause+
                                "default" ("$" VarName)? "return" ExprSingle
[53]  CaseClause            ::= "case" ("$" VarName "as")? SequenceType
                                "return" ExprSingle
[54]  IfExpr                ::= <"if" "("> Expr ")" "then" ExprSingle "else"
                                ExprSingle
[55]  OrExpr                ::= AndExpr ( "or" AndExpr )*
[56]  AndExpr               ::= InstanceofExpr ( "and" InstanceofExpr )*
[57]  InstanceofExpr        ::= TreatExpr ( <"instance" "of"> SequenceType
                                )?
[58]  TreatExpr             ::= CastableExpr ( <"treat" "as"> SequenceType
                                )?
[59]  CastableExpr          ::= CastExpr ( <"castable" "as"> SingleType )?
[60]  CastExpr              ::= ComparisonExpr ( <"cast" "as"> SingleType )?
[61]  ComparisonExpr        ::= RangeExpr ( (ValueComp
                                | GeneralComp
                                | NodeComp) RangeExpr )?
[62]  RangeExpr             ::= AdditiveExpr ( "to" AdditiveExpr )?
[63]  AdditiveExpr          ::= MultiplicativeExpr ( ("+" | "-")
                                MultiplicativeExpr )*
[64]  MultiplicativeExpr    ::= UnaryExpr ( ("*" | "div" | "idiv" | "mod")
                                UnaryExpr )*
[65]  UnaryExpr             ::= ("-" | "+")* UnionExpr
[66]  UnionExpr             ::= IntersectExceptExpr ( ("union" | "|")
                                IntersectExceptExpr )*
[67]  IntersectExceptExpr   ::= ValueExpr ( ("intersect" | "except")
                                ValueExpr )*
[68]  ValueExpr             ::= ValidateExpr | PathExpr
[69]  PathExpr              ::= ("/" RelativePathExpr?)
                                | ("//" RelativePathExpr)
                                | RelativePathExpr
[70]  RelativePathExpr      ::= StepExpr (("/" | "//") StepExpr)*
[71]  StepExpr              ::= AxisStep | FilterStep
[72]  AxisStep              ::= (ForwardStep | ReverseStep) Predicates
[73]  FilterStep            ::= PrimaryExpr Predicates
[74]  ContextItemExpr       ::= "."
[75]  PrimaryExpr           ::= Literal | VarRef | ParenthesizedExpr |
                                ContextItemExpr | FunctionCall | Constructor
[76]  VarRef                ::= "$" VarName
[77]  Predicates            ::= ("[" Expr "]")*
[78]  ValidateExpr          ::= (<"validate" "{"> | (<"validate" "global">
                                "{") | (<"validate" "context">
                                SchemaContextLoc "{") | (<"validate"
```

```
                                  SchemaMode> SchemaContext? "{")) Expr "}"
[79]  SchemaContext          ::= ("context" SchemaContextLoc) | "global"
[80]  Constructor            ::= DirElemConstructor
                                 | ComputedConstructor
                                 | XmlComment
                                 | XmlPI
                                 | CdataSection
[81]  ComputedConstructor    ::= CompElemConstructor
                                 | CompAttrConstructor
                                 | CompDocConstructor
                                 | CompTextConstructor
                                 | CompXmlPI
                                 | CompXmlComment
                                 | CompNSConstructor
[82]  GeneralComp            ::= "=" | "!=" | "<" | "<=" | ">" | ">="
[83]  ValueComp              ::= "eq" | "ne" | "lt" | "le" | "gt" | "ge"
[84]  NodeComp               ::= "is" | "<<" | ">>"
[85]  ForwardStep            ::= (ForwardAxis NodeTest) | AbbrevForwardStep
[86]  ReverseStep            ::= (ReverseAxis NodeTest) | AbbrevReverseStep
[87]  AbbrevForwardStep      ::= "@"? NodeTest
[88]  AbbrevReverseStep      ::= ".."
[89]  ForwardAxis            ::= <"child" "::">
                                 | <"descendant" "::">
                                 | <"attribute" "::">
                                 | <"self" "::">
                                 | <"descendant-or-self" "::">
                                 | <"following-sibling" "::">
                                 | <"following" "::">
[90]  ReverseAxis            ::= <"parent" "::">
                                 | <"ancestor" "::">
                                 | <"preceding-sibling" "::">
                                 | <"preceding" "::">
                                 | <"ancestor-or-self" "::">
[91]  NodeTest               ::= KindTest | NameTest
[92]  NameTest               ::= QName | Wildcard
[93]  Wildcard               ::= "*"
                                 | <NCName ":" "*">
                                 | <"*" ":" NCName>
[94]  Literal                ::= NumericLiteral | StringLiteral
[95]  NumericLiteral         ::= IntegerLiteral | DecimalLiteral |
                                 DoubleLiteral
[96]  ParenthesizedExpr      ::= "(" Expr? ")"
[97]  FunctionCall           ::= <QName "("> (ExprSingle ("," ExprSingle)*)?
                                 ")"
[98]  DirElemConstructor     ::= "<" QName AttributeList ("/>" | (">"
                                 ElementContent* "</" QName S? ">"))
[99]  CompDocConstructor     ::= <"document" "{"> Expr "}"
[100] CompElemConstructor    ::= (<"element" QName "{"> | (<"element" "{">
                                 Expr "}" "{")) Expr? "}"
```

```
[101] CompNSConstructor     ::= <"namespace" NCName "{"> Expr "}"
[102] CompAttrConstructor   ::= (<"attribute" QName "{"> | (<"attribute"
                                "{"> Expr "}" "{")) Expr? "}"
[103] CompXmlPI             ::= (<"processing-instruction" NCName "{"> |
                                (<"processing-instruction" "{"> Expr "}"
                                "{")) Expr? "}"
[104] CompXmlComment        ::= <"comment" "{"> Expr "}"
[105] CompTextConstructor   ::= <"text" "{"> Expr? "}"
[106] CdataSection          ::= "<![CDATA[" Char* "]]>"
[107] XmlPI                 ::= "<?" PITarget Char* "?>"
[108] XmlComment            ::= "<!--" Char* "-->"
[109] ElementContent        ::= ElementContentChar
                                | "{{"
                                | "}}"
                                | DirElemConstructor
                                | EnclosedExpr
                                | CdataSection
                                | CharRef
                                | PredefinedEntityRef
                                | XmlComment
                                | XmlPI
[110] AttributeList         ::= (S (QName S? "=" S? AttributeValue)?)*
[111] AttributeValue        ::= ('"' (EscapeQuot | QuotAttrValueContent)*
                                '"')
                                | ("'" (EscapeApos | AposAttrValueContent)*
                                "'")
[112] QuotAttrValueContent  ::= QuotAttContentChar
                                | CharRef
                                | "{{"
                                | "}}"
                                | EnclosedExpr
                                | PredefinedEntityRef
[113] AposAttrValueContent  ::= AposAttContentChar
                                | CharRef
                                | "{{"
                                | "}}"
                                | EnclosedExpr
                                | PredefinedEntityRef
[114] EnclosedExpr          ::= "{" Expr "}"
[115] XMLSpaceDecl          ::= <"declare" "xmlspace"> ("preserve" |
                                "strip")
[116] DefaultCollationDecl  ::= <"declare" "default" "collation">
                                StringLiteral
[117] BaseURIDecl           ::= <"declare" "base-uri"> StringLiteral
[118] NamespaceDecl         ::= <"declare" "namespace"> NCName "="
                                StringLiteral
[119] DefaultNamespaceDecl  ::= (<"declare" "default" "element"> |
                                <"declare" "default" "function">)
                                "namespace" StringLiteral
[120] FunctionDecl          ::= <"declare" "function"> <QName "(">
```

```
                                ParamList? (")" | (<")" "as"> SequenceType))
                                (EnclosedExpr | "external")
[121] ParamList             ::= Param ("," Param)*
[122] Param                 ::= "$" VarName TypeDeclaration?
[123] TypeDeclaration       ::= "as" SequenceType
[124] SingleType            ::= AtomicType "?"?
[125] SequenceType          ::= (ItemType OccurrenceIndicator?)
                                | <"empty" "(" ")">
[126] AtomicType            ::= QName
[127] ItemType              ::= AtomicType | KindTest | <"item" "(" ")">
[128] KindTest              ::= DocumentTest
                                | ElementTest
                                | AttributeTest
                                | PITest
                                | CommentTest
                                | TextTest
                                | AnyKindTest
[129] ElementTest           ::= <"element" "("> ((SchemaContextPath
                                ElementName)
                                | (ElementNameOrWildcard ","
                                TypeNameOrWildcard "nillable"?)?))? ")"
[130] AttributeTest         ::= <"attribute" "("> ((SchemaContextPath
                                AttributeName)
                                | (AttribNameOrWildcard ","
                                TypeNameOrWildcard)?))? ")"
[131] ElementName           ::= QName
[132] AttributeName         ::= QName
[133] TypeName              ::= QName
[134] ElementNameOrWildcard ::= ElementName | "*"
[135] AttribNameOrWildcard  ::= AttributeName | "*"
[136] TypeNameOrWildcard    ::= TypeName | "*"
[137] PITest                ::= <"processing-instruction" "("> (NCName |
                                StringLiteral)? ")"
[138] DocumentTest          ::= <"document-node" "("> ElementTest? ")"
[139] CommentTest           ::= <"comment" "("> ")"
[140] TextTest              ::= <"text" "("> ")"
[141] AnyKindTest           ::= <"node" "("> ")"
[142] SchemaContextPath     ::= <SchemaGlobalContext "/"> <SchemaContextStep
                                "/">*
[143] SchemaContextLoc      ::= (SchemaContextPath? QName) |
                                SchemaGlobalTypeName
[144] OccurrenceIndicator   ::= "?" | "*" | "+"
[145] ValidationDecl        ::= <"declare" "validation"> SchemaMode
[146] SchemaImport          ::= <"import" "schema"> SchemaPrefix?
                                StringLiteral <"at" StringLiteral>?
[147] SchemaPrefix          ::= ("namespace" NCName "=") | (<"default"
                                "element"> "namespace")
```