XMLGet -- User Documentation

by Neil Ferguson (neilferguson@mail.com)

Contents

1. User Documentation

1.1 Starting the Query Engine

Note : XMLGet requires a Java Runtime Environment (version 1.2 or higher) to be installed.

1. The file “xmlget.jar” should be added to the classpath environment variable:

Under MS Windows:

At the command prompt type:

```
set classpath=%classpath%;c:\xmlget.jar
```

Assuming that the 'xmlget.jar' file is held in c:\ .

Under Unix:

```
CLASSPATH=/xmlget.jar
export CLASSPATH
```

Assuming that the 'xmlget.jar' file is held in the root directory of the files system.

2. The appropriate query engine should be started. There are two different engines and you should choose the most suitable one for your needs:

Standalone Query Engine:

This engine does not provide any support for a client/server approach. All commands are read from standard input and all output is displayed on standard output. This is most

suitable for when only one user on a local machine needs to have access to the query engine. To start it type:

```
java xmlget.server.StandaloneQuery
```

Network Query Engine

This engine enables clients operating on remote machines to open dedicated connections to the machine on which the query engine (the server) is running. This suitable for when queries need to be executed remotely using Java-based clients, and high performance is required. This engine takes one command-line argument (the port number that it should be started on) and can be executed as follows:

```
java xmlget.server.NetworkQuery <port number>
```

The port number can generally be anything between 1029 and 9999, provided it is not already in use.

If a large number of documents are to be held in the system at one time you may wish to set the initial amount of memory that the Java virtual machine has, and the maximum amount that it can have. These can be set by starting Java with the '-Xms' and '-Xmx' options.

1.2 Query Engine Commands

When the query engine is started the following message will appear:

```
Query engine ready. Please enter a command…
```

The query engine is now ready to accept commands, either via. a network if using NetworkQuery, or in to the standard input of the computer that is running StandaloneQuery. Two commands are available: ‘documents’ and ‘query’. The

‘documents’ command specifies the documents that are to be parsed and added to the system, and takes a list of document URI’s as its parameter. Once the ‘documents’ command has been executed, and the specified documents have been parsed, the ‘query’ command can be executed. The ‘query’ command takes an XMLGet query as its parameter and executes this query on the documents in the system. For example, to parse the documents ‘./test.xml’ and ‘http://www.mi5.gov.uk/topsecret/doubleagents.xml’, and then execute the query ‘//name’ on them type:

```
documents=test.xml http://www.mi5.gov.uk/topsecret/doubleagents.xml
query=//name
```

If an error is not generated, the results would be returned (either over the network, or to standard output) in the following XML format:

```
<?xml version=\"1.0\" encoding=\"UTF-8\"?>
<results document="./test.xml">
results…
</results>
<results document="http://www.mi5.gov.uk/topsecret/doubleagents.xml">
results…
</results>
```

1.3 Basic Syntax Tutorial

The basic query syntax is almost identical to the core part of the ‘XML Query Language’ (XQL), as described in the XQL FAQ. Much of it is also similar to the XML Path Language (Xpath). It is recommended, however, that you read this section even if you are familiar with these technologies, as it clarifies some issues raised by the XQL specification and introduces concepts which will be helpful in understanding later sections.

Familiarity with XML is assumed; it may be helpful to imagine an XML document as a tree-type structure with each element being a node in the tree. All of the examples in this section refer to the test XML document given in appendix A.

1.3.1 Simple Path Queries

XMLGet queries are structurally similar to the Unix or MS Dos directory navigation syntax, except instead of each part of the path representing a file in the file-system it represents an element in an XML document tree. For example, the following Unix path structure represents the 'email.txt' file within the 'name' directory:

```
name/email.txt
```

Similarly, the following XMLGet query represents all ‘email’ elements within the ‘name’ element:

```
name/email
```

The Path Operator

The path operator ‘/’ is used to indicate one type of element as a child element of another one. It can be thought of as giving ‘context’ to groups of elements; for example, ‘x’ elements within ‘y’ elements will be contextually different to ‘x’ elements within ‘z’ elements.

Example -- To find all ‘customer’ elements that are child elements of ‘custorders’ elements:

```
custorders/customer
```

The above query, when used on its own, would return the tag-names, attribute values, and textual content of all ‘customer’ elements that are child elements of ‘custorders’ elements. It could also be used to specify the ‘current context’ when used within a more complex query. The concept of ‘current context’ will become clearer when we look at Filters and Subscripts in later sections.

The path operator can also be used at the start of path structures to indicate that the query should search from the root element of the document instead of the current context.

Example -- To find all ‘customer’ elements within ‘custorders’ elements starting at the root of the document:

```
/custorders/customer
```

The Recursive Descent Operator

The recursive descent operator ‘//’ is used to indicate one type of element at any level within another type. Whereas the normal path operator ‘/’ only refers to those elements that are directly below another element in the XML document tree (its "children"), the recursive descent operator refers to elements anywhere below it (its "ancestors").

Example -- To find ‘fore’ elements anywhere within ‘customer’ elements, within ‘custorders’ elements:

```
custorders/customer//fore
```

For the example data given in appendix A. the above query would be the equivalent to:

```
custorders/customer/name/fore
```

The Dot Slash Operator

The dot slash operator ‘./’ is used to specify that the query should execute from the current context rather than the root of the document. Since this is assumed anyway, this operator is only needed under one condition: where a recursive descent operator is used at the start of the query, and it should execute from the current context rather than the root.

Example -- To find all ‘name’ elements within the current context:

```
./name
```

This is equivalent to:

```
name
```

Example -- To find all ‘fore’ elements at any level, starting at the current context:

```
.//fore
```

This is different from the following, which finds all ‘fore’ elements anywhere within the document:

```
//fore
```

The Wildcard Operator

The Wildcard Operator ‘*’ is used to specify ‘any’ element, rather than particular named ones.

Example -- To find all the child elements of ‘customer’ elements:

```
customer/*
```

Example -- To find all the elements in a document:

```
//*
```

XMLGet differs from other implementations of XQL, such as the ‘GMD IPSI XQL Engine’, in that it returns only the tag-names, attributes, and textual content of the elements specified by queries, rather than their descendant elements as well; this can provide significant performance improvements, especially with ‘deep’, complex documents. The user can, however, specify that they want descendant elements to be returned by putting ‘//*’ at the end of queries.

Example -- To find all ‘customer’ elements and their descendants:

```
custorders/customer//*
```

1.3.2 Queries with Filters

Filters can be used to apply constraints to the elements within the query, and are analogous to the SQL ‘WHERE’ clause. Each filter must evaluate to a Boolean (true or false), and is tested for each element in the group to which it applies.

Filters with Comparisons

Perhaps the most useful part of a filter is that which enables one value to be compared with another one within the document. The textual value of both elements and attributes can be compared; if an element is being compared, sections of textual content that are separated by other elements, processing instructions etc. in the source document, will be separated by a white-space character ‘ ‘ for comparison purposes.

Simple Element Comparisons

The simplest comparison operator is the ‘equals’ sign ‘=’. This is used to test whether the value on the left (the left operand) is equal to the value on the right (the right operand). Boolean ‘true’ is returned if so.
Example -- To find all ‘name’ elements where the textual content of the ‘fore’ child element is equal to ‘Bob’:

```
custorders/customer/name[fore = ‘Bob’]
```

The left-hand operand in the comparison should be a sub-query. This query must, however, start from the current context and not from the root of the document (i.e. it must be a ‘relative’ query rather than an ‘absolute’ query); this is because it would be pointless to have filters where the item being compared was not within the element that the filter applied to.

The right-hand operand can be either a textual value enclosed in quotes[1] (as in the above example), or a sub-query. In this case queries do not have to start from the current context (i.e. they can be 'relative' or 'absolute').

Example -- To find all 'customer' elements where any of the 'fore' elements within 'name' elements are equal to the 'fax' element within the 'customer' element:

```
custorders/customer[name/fore = /custorders/customer/fax]
```

The left-hand operand can have multiple values (i.e. it can be a "vector"), and in the above example each 'name' element could have multiple 'fore' elements. The right-hand operand should only have a single value (i.e. it should be a 'scalar'), and in the above example there should only be one 'customer' element containing a 'fax' element, and only one fax element should be contained within it. If elements do contain multiple instances of another element, it is possible to specify which ones are used in the comparison through the use of 'Subscripts'.

Comparisons with Attributes

The left and right operands can also represent attribute values through the use of the '@' symbol followed by the name of the attribute.

Example -- To find all of the 'customer' elements where the 'id' attribute equals the value '101'.

```
custorders/customer[@id = '101']
```

Attributes can also be used at the end of path structures. They cannot, however, be used within path structures, as XML attributes cannot have child elements.

Example -- To find all of the 'custorders' elements with 'customer' child elements whose 'id' attribute is equal to '775':

```
custorders[customer/@id = '775']
```

The wildcard symbol '*' can also be used to represent any of the attributes of a given element.

Example -- To find all of the 'name' elements with attribute values equal to '232':

```
custorders/customer/name[@* = '232']
```

Multiplicity

If the left-hand operand in a comparison is expected to have multiple values, it is possible to specify whether all of them, or just any of them, must match the conditions of the comparison for the result to be true. This is achieved through the use of the 'all' and 'any' prefixes. If no prefix is given, 'any' is assumed.

Example -- To find all the 'customer' elements where all of the 'order' child elements are equal to 'Lawnmower'.

```
custorders/customer/[$all$ order = 'lawnmower']
```

Example -- To find all the 'customer' elements where any of the 'order' child elements are equal to 'Lawnmower'.

```
custorders/customer/[$any$ order = 'lawnmower']
```

Since the 'any' prefix is assumed, the above example is equivalent to:

```
custorders/customer/[order = 'lawnmower']
```

[1] Unlike XQL, the right-hand operand cannot be a numerical value without quotes. To compare numerical values use the numerical comparison operators detailed in section 1.4.1.

Basic Comparison Operators

XMLGet provides a range of basic operators for use in comparisons. These are as follows:

a = b	Returns true if a is lexically equal to b[2]
a != b	Returns true if a is lexically not equal to b
a < b	Returns true if a is lexically less than b
a > b	Returns true if a is lexically greater than b
a <= b	Returns true if a is lexically less than or equal to b
a >= b	Returns true if a is lexically greater than or equal to b
a ieq b	Case insensitive version of '='
a ine b	Case insensitive version of '!='
a ilt b	Case insensitive version of '<'
a igt b	Case insensitive version of '>'
a ile b	Case insensitive version of '<='
a ige b	Case insensitive version of '>='

There are also a number of other comparison operators that can be used for making numerical (as opposed to lexical) comparisons, and for comparing sections of strings. For further details see section 1.4.1

Filters without Comparisons

If an element or attribute is used on its own in a filter, without a corresponding comparison operand or right-hand operator, the result is 'true' if that particular element or attribute actually exists. This can be thought of as a 'there-exists-a' method, and anything that is valid for a left-hand operand is valid here.

Example -- To find all ‘name’ elements that have ‘fax’ child elements:

```
custorders/customer/name[fax]
```

Example -- To find all ‘customer’ elements that have ‘order’ child elements with ‘id’ attributes.

```
custorders/customer[order/@code]
```

Boolean Expressions

XMLGet provides support for a boolean expressions within filters.

AND Expression

The ‘AND’ boolean expression is represented in XMLGet queries by ‘and’. In the expression ‘a and b’, where a and b are either comparisons or ‘there-exists-a’ methods, , the expression will evaluate to ‘true’ if both a and b evaluate to true. Otherwise it will evaluate to false.

Example -- To find all ‘name’ elements where the ‘fore’ child element and the ‘sur’ child element are equal to ‘Bob’ and ‘Jones’ respectively:

```
custorders/customer/name[fore = ‘Bob’ $and$ sur = ‘Jones’]
```

Example -- To find all ‘customer’ elements that have at least one ‘order’ child element, and which have a ‘email’ child element equal to ‘bob@yahoo.com’:

```
custorders/customer[order $and$ email = ‘bob@yahoo.com’]
```

OR Expression

---

[2] Lexical comparison means that the two values will be compared as if they are text strings, even if they can be represented as numbers. For example, ‘1’ is lexically different from ‘001’, but numerically equal. For a full explanation, see section 1.4.1.

The ‘OR’ boolean expression is represented in XMLGet queries by ‘or’. In the expression ‘a or b’, where a and b are either comparisons or ‘there-exists-a’ methods, the expression will evaluate to ‘true’ if either a or b, or both a and b, evaluate to true. Otherwise it will evaluate to false.

Example -- To find all the ‘customer’ elements where the ‘fore’ child element is equal to ‘Robson’ or the ‘sur’ child element is equal to ‘Robson’.

```
custorders/customer[fore = ‘Robson’ $or$ sur = ‘Robson’]
```

Example -- To find all the ‘customer’ elements that have either ‘order’ or ‘fax’ child elements.

```
custorders/customer[order $or$ fax]
```

NOT Expression

The ‘NOT’ boolean expression is represented in XMLGet queries by ‘not’. In the expression ‘not a’, where a is either a comparison or a ‘there-exists-a’ method, the expression will evaluate to ‘true’ if a evaluates to ‘false’. Otherwise it will evaluate to false.

Example -- To find all the ‘customer’ elements where the ‘fore’ child element is not equal to ‘Bob’

```
custorders/customer[$not$ fore = ‘Bob’]
```

Example -- To find all the ‘customer’ elements that do not have ‘fax’ child elements.

```
custorders/customer[$not$ fax]
```

Ordering Boolean Expressions

When combining boolean expressions parenthesis '(' and ')' can be used to determine the order in which they should be evaluated. Expressions inside parenthesis will always be evaluated first; otherwise expressions will be evaluated from left to right.

Example -- To find all 'customer' elements that do not have 'order' child elements, and do not have 'sur' child elements equal to 'Smith'.

```
custorders/customer[$not$(order $and$ sur = 'Smith')]
```

This can be compared to the following, which will find all 'customer' elements that do not have 'order' child elements, and **do** have 'sur' elements equal to 'Smith':

```
custorders/customer[$not$ order $and$ sur = 'Smith']
```

1.3.4 Subscripts

Subscripts allow a particular element to be selected according to its position in a set, and they use a similar notation to filters to specify the index of that element.

Example -- To find the first 'email' element within a 'name' element:

```
name/email[1]
```

1.3.5 Union and Intersection

XMLGet allows two set operations: union and intersection, to be used in queries. These are represented by '$union$' and '$intersect$' respectively.

Union

Union of Queries

The union operator can be used to combine the results of two queries. In the expression ‘a $union$ b’ where both a and b are XMLGet queries, a collection containing the results of both queries will be returned.

Example -- To find all of the ‘fore’ elements equal to ‘Bob’, plus all of the ‘order’ elements with “id” attributes equal to ‘101’:

```
custorders/customer/name/fore[.=’Bob’] $union$
custorders/customer/order[@id = ‘101’]
```

Union within Queries

The union operator can also be used to combine groups of elements within queries. In the expression ‘(a $union b)/c’, where a, b are relative path structures and c is a group of elements, all of the elements of type c that are child elements of those represented by a or b will be returned.

Example -- To find all ‘countrycode’ elements within ‘fax’ and ‘telephone’ elements:

```
custorders/customer/(fax $union$ telephone)/countrycode
```

Example -- To find all ‘email’ elements and ‘postcode’ elements within ‘address’ elements, all within customer elements:

```
custorders/customer/(address/postcode $union$ email)
```

Note that when unions are performed within queries, parenthesis **must** be used to avoid ambiguity. If the parenthesis were removed from the above example, it would be taken to mean the union of the query ‘custorders/customer/address’ with the separate query ‘email’.

Intersection

Intersection between Queries

The intersection operator can be used to return a collection of those elements from two different queries that have textual content in common. In the expression ‘a $intersect$ b’, where a and b are XMLGet queries, a collection of elements will be returned containing elements from the results of both a and b. The textual content of those elements from the results of query a must match the textual content of at least one element from the results of b.

Example -- To find all of the ‘postcode’ elements and ‘itemid’ elements where every ‘postcode’ element is equal to at least one ‘itemid’ element.

```
custorders/customer/address/postcode $intersect$
custorders/customer/order/itemid
```

Intersection within Queries

The intersection operator can also be used to find elements that intersect within queries. In the expression ‘a/(b $intersect$ c)’, where a, b, and c are groups of elements, a collection will be returned containing elements of types b and c. The textual content of those elements of type b must match the textual content of at least one element of type c, and all the elements in the collection must be child elements of a.

Example -- To find all ‘fore’ and ‘sur’ elements where each ‘fore’ element is equal to at least one ‘sur’ element (i.e. somebody has the same surname and forename):

```
custorders/customer/name/(fore $intersect$ sur)
```

Example -- To find all ‘number’ elements within ‘fax’ or ‘telephone’ elements, where each ‘fax’ element is equal to at least one ‘telephone’ element.

```
custorders/customer/(fax $intersect$ telephone)/number
```

As with unions, when intersections are used within queries, parenthesis **must** be used to avoid ambiguity.

Set Operations within Filters.

Set operations can also be used within filters. They can be performed either within sub-queries, or between two sub-queries on the left-hand side of a comparison. Set operations cannot be performed between sub-queries on the right-hand side of a comparison, as they will always return multiple values, whereas the right hand side of a comparison can only have a single value.

Set operations **can** be performed on right-hand operands when a special type of comparison called a 'group comparison' is being used; for further details see section 1.4.1

Example -- To find all 'customer' elements whose 'name' child elements have at least one 'fore' and 'sur' child element with identical textual content, and at least one of these elements is equal to 'Bob' (i.e. somebody is called "Bob Bob"):

```
custorders/customer[name/(fore $intersect$ sur) = 'Bob')]
```

Example -- To find all 'customer' elements where any of the 'id' attributes or the 'countrycode' child elements of the 'telephone' child elements are equal to '667':

```
custorders/customer[telephone/countrycode $union$ @id = '667']
```

Ordering Set Operations

Like boolean expressions, the order in which set operations are performed can be determined by using parenthesis. Operations within parenthesis will always be performed before those outside.

Example -- To perform an ‘intersection’ operation between ‘fax/number’ elements and ‘telephone/number’ elements and then a ‘union’ operation between these and ‘customer/email’ elements:

```
custorders/customer/email $union$
(custorders/customer/fax/number $intersect$
custorders/customer/telephone/number)
```

1.4 Advanced Syntax Tutorial

1.4.1 Advanced Comparisons

Numerical Comparison Operators

XMLGet has a number of operators for comparison of numerical values. Since ‘1’ is lexically equal to ‘001’ but numerically not equal, and ‘10’ is lexically less than ‘2’ but numerically greater, this is a very important feature. Numerical comparison operators are identical to their lexical counterparts, except they have periods ‘.’ on either side (e.g. ‘.=.’). This is similar to the syntax of xtract.

When a numerical comparison operator is used in XMLGet, the query engine will attempt to convert both the left and right operands in the comparison to numerical values. If the conversion fails, a lexical comparison will be performed.

Example -- To find all those ‘customer’ elements that have numerical ‘id’ attributes that are numerically less than 100.

```
//customer[@id .<. ‘100’]
```

XMLGet’s method of numerical comparison differs slightly from XQL, which uses quotes (‘’) to specify values for lexical comparison, and values without quotes to imply numerical comparison. This is fine for queries like ‘//customer[@id < 100]’, but since XML does not support typecasting it is impossible to determine if a query like

‘//customer[telephone/number = fax/number]’ is supposed to use lexical or numerical comparison.

Example -- To find all those ‘customer’ elements where a ‘number’ child element of a ‘telephone’ child element is numerically less than the ‘number’ child element of the ‘fax’ child element:

```
//customer[telephone/number .<. fax/number]
```

The full set of numerical comparison operators is as follows:

a .=. b	Returns true if a is numerically equal to b
a .!=. b	Returns true if a is numerically not equal to b
a .<. b	Returns true if a is numerically less than b
a .>. b	Returns true if a is numerically greater than b
a .<=. b	Returns true if a is numerically less than or equal to b
a .>=. b	Returns true if a is numerically greater than or equal to b

Non-Exact Comparison Operators

Often it is desirable to make comparisons based on specific parts of the textual content of an element or attribute, rather than the entire content. The non-exact comparison operators are therefore an important part of XMLGet’s query language.

Sub-string Operator

The sub-string operator (‘$contains$’) can be used to find out whether the text string represented by the right-hand operand in a comparison is a sub-string of the left-hand operand. The syntax for sub-string comparisons is exactly the same as for any other comparison.

Example -- To find all ‘fore’ elements containing the letter ‘a’:

```
//name/fore[. $contains$ ‘a’]
```

There is also a case-insensitive version of the sub-string operator; this is represented by ‘$icontains$’.

Example -- To find all ‘customer’ elements with ‘email’ child elements containing ‘Yahoo’, ‘YAHOO’, ‘yahoo’ etc.

```
custorders/customer[email $contains$ ‘yahoo’]
```

Regular Expression Matching

XMLGet supports regular expression matching through the ‘$regmatch$’ comparison operator. In the expression ‘a $regmatch$ b’, where a represents a set of elements or attributes and b represents a regular expression, the expression will evaluate to true if the textual content of any element or attribute within a matches the regular expression b.

Example -- To find all ‘postcode’ elements whose textual content contains three digits.

```
//customer/address/postcode[. $regmatch$ ‘\{d}3’]
```

As in other comparisons, the right-hand operand does not have to be a value enclosed in quotes, but can be a sub-query, provided the sub-query returns a single value that represents a valid regular expression.

For full details of supported regular expression syntax see the GNU Regexp syntax details.

Set Comparison Operators

As explained in section 1.3.2, normal comparisons require the right-hand operand to represent only a single value. With simple queries this is acceptable, but with more

complex ones, perhaps ones with many sub-queries, we need to be able to do set comparisons; those readers familiar with SQL may wish to imagine the usefulness of that language without sub-queries being able to return multiple values.

Analogous to the SQL 'IN' operator, XMLGet allows set comparisons to be performed using the 'in' operator. In the expression 'a in b', where a and b represent groups of elements or attributes, the expression will evaluate to 'true' if the textual content of at least one element from group a is equal to at least one element from group b. Otherwise it will evaluate to 'false'.

Example -- To find all 'name' child elements of 'country' elements whose sibling element 'code' matches any of the 'countrycode' elements within 'customer' elements (i.e. to find all the names of countries that customers have telephone or fax numbers in):

```
//country[code $in$ //customer//countrycode]/name
```

The case-insensitive version of the set comparison operator is 'iin'.

1.4.2 Display Filters

One major weakness of XQL as a query language is the fact that it can only return elements from one level of a document at a time[3]. For example, if we wanted to find the email addresses of all customers, XQL could only return a list of 'email' elements on their own, without returning the 'customer' elements they related to. Those readers familiar with SQL may wish to imagine the usefulness of that language if they were only able to put a single field name after the 'SELECT' operator. XQL **can** return elements from different levels of a document if queries are joined by the '$union$' operator, but this can be quite inefficient and it is impossible to tell which elements relate to each other. For example, from the results of the query 'custorders/customer $union$

[3] XQL can return elements from different levels of a document if a recursive descent operator (see section 1.3.1) is used; the elements have to all have the same name, however.

custorders/customer/email' we cannot determine which 'email' elements are children of which 'customer' elements.

XMLGet has two display filters. The '{content}' filter will display the textual content of a particular element, along with its tag name and attributes, and the '{name}' filter will just display an element's tag name and attributes. Display filters can be used after each element name in a query; if filters or subscripts are used, they should be placed after the filter or subscript. It is syntactically valid to use display filters in sub-queries, but they will just be ignored if they are used.

Example -- To find all 'customer' elements and their associated 'email' child elements:

```
custorders/customer{content}/email
```

Example -- To find the names of all elements whose textual content contains the letter 'a':

```
//*[. $contains$ 'a']{name}
```

Example -- To find all the tag-names (including attribute values) of all 'customer' elements along with their 'email' child elements, and all those elements within them that have 'number' child elements containing '0'.

```
custorders/customer{name}/(email $union$ .//*[number $contains$
'0'])
```

In the above example, the 'email' elements and other elements within 'customer' will be returned from the query as child elements of 'customer' (even if they are not child elements in the document).

The elements at the end of every query (after the last path operator '/') will always be displayed as if they have an associated {content} display filter, unless another one is explicitly specified.

Although it is syntactically correct to use a display filter in a sub-query, all such display filters will be ignored.

Formal Query Grammar

The formal grammar is given in BNF (Backus-Naur Form) notation. It borrows the `Name`, `Char` and `Digit` productions from the W3C XML 1.0 Recommendation.

```
Query                   ::=   SetOperation

SetOperation            ::=   Path | SetOperation SetOp SetOperation

SetOp                   ::=   '$union$' | '$intersect$'

BooleanExpression       ::=   Comparison |
                              BooleanExpression BooleanOp
                              BooleanExpression |
                              '$not$' BooleanExpression |
                              '(' BooleanExpression ')'

BooleanOp               ::=   '$and$' | '$or$'

Comparison              ::=   Lvalue |
                              MultiplicityOp* LValueSetOperation
                              ComparisonOp Rvalue |

MultiplicityOp          ::=   '$any$' | '$all$'

ComparisonOp            ::=   '=' | '<' | '>' | '<=' | '>=' | '$ieq$' |
                              '$ine$' | '$ilt$' | '$igt$' | '$le$' |
                              '$ge$' | '.=.' | '.<.' | '.>.' | '.<=.' |
                              '.>=.' | '$contains$' | '$icontains$' |
                              '$regmatch$'

SetComparisonOp         ::=   '$in$' | '$iin$' | '$.in.$'

LValueSetOperation      ::=   Lvalue |
                              LvalueSetOperation SetOp
LvalueSetOperation |
                              '(' LvalueSetOperation ')'

LValue                  ::=   RelativePath | Attribute |
                              RelativePath '/' Attribute

Attribute               ::=   '@' (* | Name)

RValue                  ::=   Path | Attribute | Path '/' Attribute |
                              ''' Char* '''
```

```
Path                    ::=   AbsolutePath | RelativePath

AbsolutePath            ::=   PathOp RelativePath

PathOp                  ::=   ‘/’ | ‘//’

RelativePath            ::=   DisplayFilter |
                              DisplayFilter PathOp RelativePath

DisplayFilter           ::=   Subscript |
                              Subscript DisplayOperation |
                              ‘(‘ NonPathSetOperation ‘)’

NonPathSetOperation     ::=   RelativePath SetOp NonPathSetOperation |
                              RelativePath SetOp RelativePath

DisplayOperation        ::=   {content} | {name}

Subscript               ::=   Filter |
                              Filter ‘[‘ Digit ‘]’ |

Filter                  ::=   RelativeTerm |
                              RelativeTerm ‘[‘ BooleanExpression ‘]’

RelativeTerm            ::=   ‘*’ | ‘.’ | Name
```