

Workshop V.: Oracle XSQL Servlet

Author: Gergely Mátéfiy

Using the earlier works by Gábor Nagypál, István Bihari, and Zoltán Hajnács

1	INTRODUCTION	65
2	XML DOCUMENT STRUCTURE.....	66
2.1	ELEMENTS AND TAGS	66
2.2	STRUCTURE	66
2.3	ATTRIBUTES	67
2.4	ENTITY REFERENCES.....	67
2.5	NAMESPACES	67
3	CREATING XML DOCUMENTS USING XSQL TEMPLATES.....	68
4	PARAMETER HANDLING IN XSQL TEMPLATES.....	69
4.1	PARAMETER HIERARCHY	69
4.2	SETTING PARAMETER VALUES	70
4.3	USING PARAMETERS	70
4.4	INSERTING PARAMETERS INTO THE XML OUTPUT	71
5	THE XSL TRANSFORMATION	72
6	XPATH EXPRESSIONS	74
7	ASSIGNING XSL STYLESHEETS TO XML DOCUMENTS.....	76
8	CONDITIONAL TESTS AND VARIABLES ON XSL STYLESHEETS.....	77
8.1	CONDITIONAL TEST.....	77
8.2	MULTIPLE CONDITIONAL TEST	78
8.3	VARIABLES	78
9	XSLT TEMPLATES	78
9.1	RECURSIVE TEMPLATE PROCESSING.....	78
9.2	REUSABLE AND NAMED TEMPLATES	80
9.3	ORGANIZING STYLESHEETS.....	80
10	DEMO APPLICATION.....	80
11	GETTING READY	85
12	REFERENCES	85
13	APPENDIX: XSQL REFERENCE.....	87
14	APPENDIX: XPATH FUNCTION REFERENCE.....	90
15	APPENDIX: XSLT REFERENCIA	91

1 Introduction

Generally, large IT systems have several interfaces – for example windows client, web interface, data link interface towards external IT system etc. Having multiple interfaces is not efficient in client-server architecture; therefore complex systems are usually built based on a *multi-level architecture*. In a multi-level architecture the *application logic layer* responsible for data element validity and the *presentation layer* in charge for the user interface management are separated and the different interfaces are using the common application layer.

For the integration of software components found in these layers originating from different manufacturers and sometimes running on different platforms a common language is required. In the past few years Extensible Markup Language (XML) became the de-facto standard of describing data elements. The XML is a platform-independent, text-based markup language that allows structured description and sharing of information and data elements. Its related standards include XML schema definition and validation, and transformation between schemas.

The XSQL servlet provided by Oracle is a software component that bridges the SQL and the XML domains: it is transforming incoming XML data packages into data manipulation SQL statements that are executed in a relational database; and it also creates outgoing XML data packages based on SQL query and operation results.

The aim of the present workshop is to give an insight into XML-based application development by using the XSQL servlet. The first chapter introduces the structure of XML documents, which is followed by the overview on the structure of XSQL templates. Later it is investigated how the XSQL output can be transformed using XSLT and XPath technologies, and finally a demo summarizes all these features described before. The present document is based on the XML 1.0, XSLT 1.0, and XPath 1.0 standards.

2 XML document structure

2.1 Elements and Tags

Similarly to HTML, XML documents are text-based files containing nested elements. Elements are marked with a start-tag and an end-tag, as it is illustrated in the following example:

```
<?xml version="1.0" encoding="ISO-8859-2"?>
<message>
  <from>Jack</from>
  <to>Jill</to>
  <body>Hello world!</body>
</message>
```

Unlike in HTML, there is no fixed tag vocabulary in XML. Instead, it is the application-dependent grammar (XML schema) that defines the tags and the nesting rules to be used in the document. According to XML terminology a document is *well-formed* if its syntax conforms to the XML syntax rules; and it is *valid* if it conforms to semantic rules – either user-defined or in an XML schema.

Tag names can begin with a letter or an underscore (“_”). The XML is case-sensitive, therefore the following two labels <From> and <from> are considered to be different.

2.2 Structure

Every XML document starts with an XML declaration element stating what XML version is in use and it might also contain character encoding:

```
<?xml version="1.0" encoding="ISO-8859-2"?>
```

A generic XML document contains a tree-based data structure. The document has exactly one root element, in the previous example the root element was <message>. For every start-tag there has to be an end-tag, and these two encapsulate the element content. Unlike the more relaxed rules of HTML, XML requires that elements must be properly nested – elements may never overlap, and so must be closed in the order opposite to which they are opened. For example the following fragment is correct in HTML, but is not a well-formed XML:

```
<b>This is bold <i> and this is bold and italic </b> but that is only italic </i>
```

A well-formed XML fragment would be:

`This is bold <i> and this is bold and italic </i><i> but that is only italic </i>`

XML provides special syntax for representing an element with empty content. Instead of writing a start-tag followed immediately by an end-tag, a document may contain an empty-element tag. For example `
</br>` is equivalent with `
`.

Just like in HTML, comments can be placed anywhere in the XML document using the following syntax: `<!-- Comment -->`. Please note that comments cannot be nested.

2.3 Attributes

According to the XML logic element properties can either be described with further elements, or with attributes – based on the rules described in the applied schema. Attributes are placed in the start-tag, for example:

`<message date='20090221'>Hello!</message>`

Attribute values must always be quoted, using single or double quotes – however, unlike in HTML quotes cannot be omitted. Each attribute name may appear only once in any single element.

2.4 Entity references

In the element contents marked with tags there might be any characters apart from the reserved `<` and `&`. These special control characters can be represented in XML using entity references – see the table below. For example the following formula is incorrect:

`<formula>a < b & b < c => a < c </formula>`

Instead, the following entity references should be used:

`<formula>a < b & b < c => a < c</formula>`

Alternatively, the special `<![CDATA ...]>` section might also be used:

`<formula><![CDATA a < b & b < c => a < c]></formula>`

Entity	&	<	>	"	'	Line break
Entity reference	&	<	>	"	'	

2.5 Namespaces

Since XML vocabularies are defined by different applications ambiguity between identically named elements or attributes might occur when merging documents originating from these applications. XML *namespaces* are used for providing uniquely named elements and attributes in an XML instance. A namespace is declared using the reserved XML attribute `xmlns`, the value of which must be an Internationalized Resource Identifier (IRI), usually a Uniform Resource Identifier (URI) reference. For example XSLT uses the `"http://www.w3.org/1999/XSL/Transform"`, while XSQL uses the `"urn:oracle-xsql"` namespace. In order to use the tags belonging to a certain namespace, a unique prefix must be defined using the special attribute: `xmlns: "prefix"="namespace URI"`. In this case, any element or attribute names that start with the prefix `"prefix:"` are considered to be in the namespace URI namespace. The usage of prefixes is illustrated by the following example:

```
<?xml version="1.0" encoding="ISO-8859-2"?>
<mail:message from="jack" to="Jill" xmlns:mail="internet:mail">
  <mail:subject>Meet</mail:subject>
  <mail:body xmlns:xhtml="http://www.w3.org/1999/xhtml">
    <xhtml:body>
      Let's meet <xhtml:b>at 6 PM</xhtml:b> in our favourite pub!
    </xhtml:body>
  </mail:body>
</mail:message>
```

3 Creating XML documents using XSQL templates

Servlets are server-side Java applications – connecting to the web server through a standard API – accepting HTTP requests and generating answers. The *Oracle XSQL* is a servlet which is embedding results of SQL operations into an XML document by processing XSQL templates.

XSQL templates are special XML files containing directives that belong to the "urn:oracle-xsql" namespace. Upon an XSQL template request the web server is forwarding the request to the XSQL servlet which executes the XSQL action elements included in the template and sends the XML format result to the output. An XSQL template might contain labels belonging to a namespace other than "urn:oracle-xsql" but these labels are not processed – they are displayed on the output without modification.

The following example illustrates the SQL template required to display the results of a simple SQL statement:

```
<?xml version="1.0" encoding="ISO-8859-2"?>
<page connection="workshop" xmlns:xsql="urn:oracle-xsql">
  <xsql:query>
    SELECT isbn, author, title FROM book ORDER BY isbn
  </xsql:query>
</page>
```

Similarly to the XML syntactical requirement the XSQL template has exactly one root element – in the previous example the root start-tag and end-tags are <page> and </page>, respectively.

The information required for creating the database connection (database URL, login name and password) is not included in the template file, but is contained by the XSQL servlet configuration. The template is only referencing (and it should reference) to a pre-configured connection by setting the connection attribute – in the second line of the above code segment the connection configured under the name workshop is used. In the same line where xsql: prefix is defined for the XSQL statements.

The **xsql:query** action element located between line 3 and 5 executes a SELECT statement and writes its result set into the resulting XML in some sort of ROWSET/ROW format. By running the example code above the result looks like the following:

```

<?xml version="1.0" encoding="ISO-8859-2"?>
<page>
<ROWSET>
  <ROW num="1">
    <ISBN>963 211 773 5</ISBN>
    <AUTHOR>Douglas Adams</AUTHOR>
    <TITLE>The Hitch Hiker's Guide to the Galaxy</TITLE>
  </ROW>
  <ROW num="2">
    <ISBN>963 10 9436 7</ISBN>
    <AUTHOR>B.W. Kernighan-R. Pike</AUTHOR>
    <TITLE>The UNIX Programming Environment</TITLE>
  </ROW>
</ROWSET>
</page>

```

The format is the so-called XSQL canonical format: the result set is marked by the label ROWSET, a particular result line is marked by the label ROW, respectively. The attribute num marks the row number. Child elements of ROW contain the values of the fields, marked by uppercase labels matching the column names. Please note, that a valid SQL column name might not always result in a valid XML label – for example COUNT(*). In these cases the programmer has to rename the column in the SQL query – in the statement write “COUNT(*) AS Total” – in order to avoid runtime error.

The execution of the **xsql:query** action element can be controlled by several attributes, most importantly by:

- null-indicator: with its value set to no the NULL value fields are not displayed on the output. With the value yes the NULL value fields are marked with an empty element tag containing the attribute NULL="TRUE". The default value of the null-indicator is no;
- skip-rows: the first „skip-rows” number of result lines are ignored;
- max-rows: specifies the maximum number of result rows requested (herein the default value -1 means all results are required).

When an XSQL action element execution results in **database error**, instead the result set an XML segment describing the error appears, as it is illustrated by the following example:

```

<xsql-error code="error code" action="invalid xsql statement">
  <statement>The content of the invalid SQL statement</statement>
  <message>Error message</message>
</xsql-error>

```

4 Parameter handling in XSQL templates

4.1 Parameter hierarchy

Since ordinary variables cannot be used in XSQL templates, XSQL action element execution can be controlled by static or dynamic – set by queries – parameters received from the web server.

Parameters are arranged in a strict hierarchy: in case more parameters exist under the same name then upon referencing the parameter the higher hierarchy level value will be considered. In the followings the parameters are listed in a decreasing hierarchical order:

1. **Page parameters:** parameters defined explicitly in the XSQL template using the `xsql:set-page-param` action element. These parameters are valid during the processing of the given XSQL page.
2. **HTTP cookies:** parameters stored in the browser that can be set by the `xsql:set-cookie` action element.
3. **HTTP session parameters:** valid only during the browser – web server interaction these parameters are stored on the web server side and can be set using the `xsql:set-session-param` action element.
4. **HTML form parameters** (obviously cannot be changed)
5. **Action element attributes:** for example `<xsql:query paramname="value">`
6. **Action element ancestor attribute:** for example `<root paramname="value">`

4.2 Setting parameter values

Page, cookie and session parameters can be set to a constant value, can be assigned another parameters' values and can also be configured dynamically, by SQL queries. The examples below illustrate three different ways of setting a page parameter:

```
<xsql:set-page-param name="par1" value="42"/>
<xsql:set-page-param name="par2" value="{@par1}"/>
<xsql:set-page-param name="par3">
  SELECT COUNT(*) FROM book
</xsql:set-page-param>
```

The session and cookie parameters defined earlier, and the HTML form parameters received from the browser are created automatically and become accessible during the XSQL template run. The new values of session and cookie parameters are becoming effective in the subsequent runs – and not immediately.

4.3 Using parameters

On the XSQL pages the parameters can be accessed using the `{@parameter name}` syntax. With this particular kind of referencing the **parameter value is substituted in text form**. In the example below the author's name and the book's title are provided by the HTML form:

```
<?xml version="1.0" encoding="ISO-8859-2"?>
<page connection="workshop" xmlns:xsql="urn:oracle-xsql">
  <xsql:query author="%" title="%" >
    SELECT isbn, author, title FROM book
    WHERE author LIKE '{@author}' and title LIKE '{@title}'
  </xsql:query>
</page>
```

This example also illustrates how default value is assigned to a form parameter. In the `xsql:query` tag two attributes: `author` and `title` are defined. In case these parameters are not provided by the form, then attribute values from a lower parameter-hierarchy level will be substituted.

Substituting texts into SQL statements creates major security issues, therefore instead of the example shown above the use of **parameter SQL queries** is recommended:

```
<?xml version="1.0" encoding="ISO-8859-2"?>
<page connection="workshop" xmlns:xsql="urn:oracle-xsql">
  <xsql:query author="%" title="%" bind-params="author title">
    SELECT isbn, author, title FROM book
    WHERE author LIKE ? and title LIKE ?
    ORDER BY isbn
  </xsql:query>
</page>
```

In SQL statements parameters are marked with '?'. In the `bind-params` attribute the XSQL parameters to be substituted are listed in the right order, and are separated by space. Herein, substitution only takes place in the DBMS after the SQL statement interpretation.

Parameters provided by the HTTP request can also be used in data manipulation SQL statements: using the **xsql:dml** action element an arbitrary SQL statement or PL/SQL block can be executed. The following example illustrates how to modify the title of a book:

```
<?xml version="1.0" encoding="ISO-8859-2"?>
<page connection="workshop" xmlns:xsql="urn:oracle-xsql">
  <xsql:dml bind-params="title isbn" commit="yes">
    UPDATE book SET title=? WHERE isbn=?
  </xsql:dml>
</page>
```

As default the `xsql:dml` action element does not commit the modifications, therefore the programmer must explicitly ask for the commit by setting the attribute `commit='yes'`. The number of modified rows is inserted into the output by `xsql:dml` in a status message:

```
<xsql-status action="xsql:dml" rows="1"/>
```

4.4 Inserting parameters into the XML output

The value of a parameter can be inserted into the XML output created by the XSQL servlet using the **<xsql:include-param name="paramname"/>** action element. The **<xsql:include-request-params/>** action enables the user to include all parameters (form, session, cookie) in the XML output.

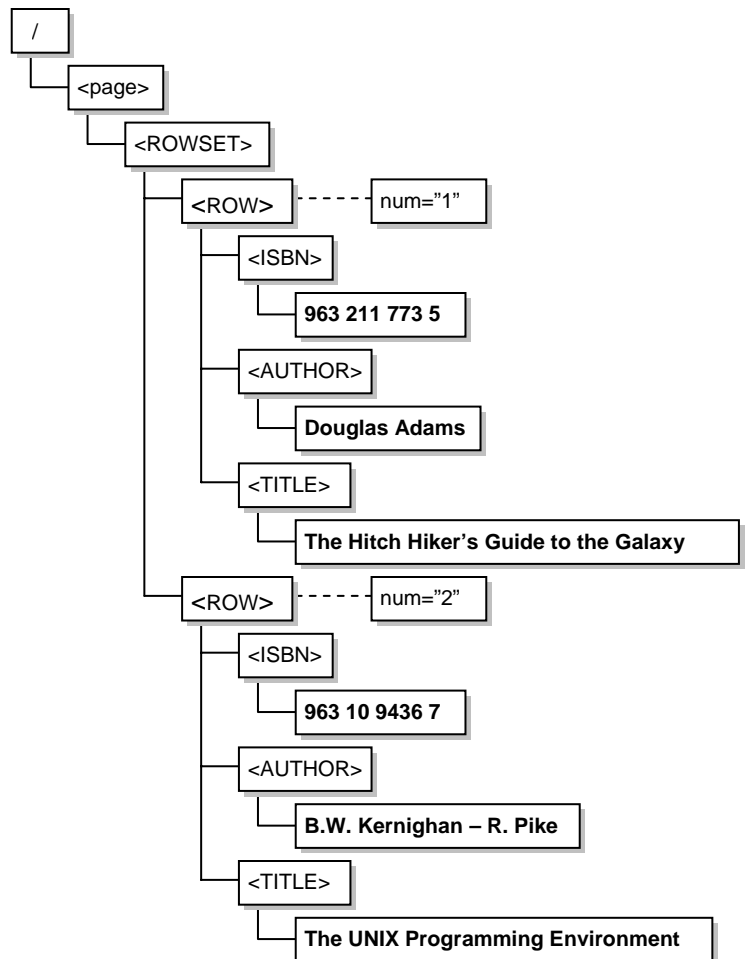
5 The XSL Transformation

Extensible Stylesheet Language Transformations (XSLT) is an XML-based language used for the transformation of tree structured representation XML documents into an output tree structure. The rules of XSL transformation are defined by *XSL stylesheets*¹. The software actually performing the XSL transformation is called the *XSLT processor*.

This figure illustrates the tree structure representation of the canonic output of the library example introduced in the previous chapters. In the tree structured representation the XML document is presented as a tree consisting of nodes. The following node types can be identified: *root node*, *element node*, *text node*, *attribute node*². In all cases the document begins with the root node representing the document itself. The root node has exactly one child node – that is `<page>` in the present example. Element nodes can have further attribute, text and element nodes as child nodes.

Please note, that the figure does not entirely match the earlier text representation, which was enhanced with line-breaks and tabs for the sake of better readability. Indenting is absolutely legal in XML syntax; however these indenting characters should be represented in the tree structure as text nodes, as well.

The XSL stylesheet itself is an XML document using statements belonging to the XSLT namespace to describe the transformation. The structure of stylesheets is demonstrated by the example below, where a list of books in canonic format is transformed into a HTML table format:



¹ Please note: XSL stylesheets are not equivalent to the CSS (Cascading Style Sheets) of the HTML standard!

² For the sake of a complete list of node types, here are the others: namespace node, processing instruction node, and comment node.


```

<?xml version="1.0" encoding="ISO-8859-2"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>
<xsl:template match="/">
<html>
  <body>
    <table>
      <xsl:for-each select="page/ROWSET/ROW">
        <tr>
          <td><xsl:value-of select="ISBN"/></td>
          <td><xsl:value-of select="AUTHOR"/></td>
          <td><xsl:value-of select="TITLE"/></td>
        </tr>
      </xsl:for-each>
    </table>
  </body>
</html>
</xsl:template>
</xsl:stylesheet>

```

The root element of the XSL stylesheet is `<xsl:stylesheet>`. A stylesheet might contain one or more templates. A template can be defined for any node of the source XML document; in this case the template matching the node defines the transformation of that particular sub-tree. Templates will be described in more details later in this document, for the sake of simplicity herein only one template is defined using the `<xsl:template>` element which applies to the root element. This example template contains both HTML tags (`<body>`, `<table>`, `<tr>`, `<td>`) and XSLT elements such as `<xsl:for-each>` and `<xsl:value-of>`. During the XSL processing tags outside the XSLT namespace (herein the HTML tags) are sent to the output unchanged, while XSLT elements are interpreted and executed.

The XSLT element **`xsl:for-each`** in the example denotes iteration: loops on the nodes selected in the `select` attribute and performs the nested statements for every sub-tree. The „page/ROWSET/ROW” expression used in the example selects the ROW elements from the XML source.

The XSLT element **`xsl:value-of`** creates a text node in the output. The content of the text node is defined by the expression provided in the `select` attribute. In the above example, the expressions “ISBN”, “AUTHOR”, “TITLE” are returning the values of the actual sub-trees – meaning the values of the requested fields.

At the end of the transformation the resulting tree is printed on the output by the XSLT processor in text form. This process is called the serialization of the result tree. The **`xsl:output`** element of the XSL stylesheet controls the serialization process. The XSLT 1.0 supports three different output methods:

- `<xsl:output method="xml"/>`: printing nodes in a well-formed XML format;
- `<xsl:output method="html"/>`: printing nodes in a HTML 4.0 compatible format, therefore for example the empty `
` tag is replaced by `
` on the output;
- `<xsl:output method="text"/>`: only nodes are printed, without the markup.

In our example the "html" method was chosen, therefore the output of the transformation will look like this:

```
<html>
  <body>
    <table>
      <tr>
        <td>963 211 773 5</td>
        <td>Douglas Adams</td>
        <td>The Hitch Hiker's Guide to the Galaxy</td>
      </tr>
      <tr>
        <td>963 10 9436 7</td>
        <td>B.W. Kernighan-R. Pike</td>
        <td>The UNIX Programming Environment</td>
      </tr>
    </table>
  </body>
</html>
```

If the stylesheet contains only one template matching the root element then the **simplified syntax** can also be used. In this case the `<xsl:stylesheet>` and `<xsl:template>` elements are omitted and the root element within the template becomes the root of the stylesheet, and the XSL namespace declaration is put into the new root. Therefore the example stylesheet with using the simplified syntax looks like this:

```
<?xml version="1.0" encoding="ISO-8859-2"?>
<html xsl:version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <body>
    ...
  </body>
</html>
```

In case of using the general syntax the default output method is "xml" while for simplified syntax – if the root is `<html>` – is "html".

6 XPath expressions

The XPath standard language (XML Path Language) is based on a tree representation of the XML document, and provides the ability to navigate around the tree, selecting nodes by a variety of criteria. The result of an XPath expression might be a set of nodes (sub-trees), numbers, strings or Boolean values.

Also in the previous XSL example several XPath expressions were used: in the `<xsl:template>` element `/` marks the root; in the `<xsl:for-each>` element the `"page/ROWSET/ROW"` expression marks the sub-trees representing the rows; while in the `<xsl:value-of>` element the expressions `"ISBN"`, `"AUTHOR"`, `"TITLE"` are selecting the nodes representing the respective fields of the actual sub-tree.

The XPath location path is similar to a path in a DOS or UNIX change directory command: it defines a path in the XML tree structure. Similarly to file systems a path can be either *absolute* or *relative*. A full path or absolute path is written in reference to the root and the path starts with the

character “/”; in a relative path the starting point is the actual node. The location path consists of location steps which are separated by “/”. The general form of a location step is³:

node test[predicate]

Predicates are optional, thus can be omitted. The node test formats are the following:

Format	Selected node(s)
name	Child node called „ name”
@name	Attribute node called „ name”
.	Self node
..	Parent node
*	Node element type child nodes
@*	Node attribute type child nodes
//	All descendants and self node
node()	Finds any node
text()	Finds a node of type text
comment()	Finds an XML comment node

Predicates in location steps are complex XPath logical expression containing functions and operators, where the expression must be satisfied before the preceding node will be matched by an XPath. Some typical expressions:

Expression	True, if ...
element	there is a node called element
element=value	the value of node element is value
@attr=value	the value of attribute attr is value
position()=n	the node is the n th element of the set
count(set)=n	the number of nodes that can be reached via the XPath is n
sum(set)=n	the sum of the values of nodes that can be reached via the XPath is n

Boolean operators "and" and "or", and a function "not()" can also be used in predicates. Instead of the [position()=n] form the simplified notation of [n] can also be used. The list of the most important functions used in XPath location paths can be found in the Appendix.

The union of two node-sets selected by two XPath expressions can be formed using the union operator “|”.

³ Herein only the abbreviated syntax is described. In the unabbreviated syntax a location step is extended by navigation direction (axis).

For the sake of better understanding hereby we include a few XPath examples which were earlier interpreted for canonic XML output:

- `/`: the root of the document (not equivalent to the root address!);
- `/page/ROWSET/ROW`: sub-trees representing all result lines of all queries;
- `/page/ROWSET[1]/ROW[1]/*`: the fields of the first result row of the first query;
- `/page/ROWSET/ROW[AUTHOR]`: result rows where the AUTHOR field is not NULL (meaning there is a node representing the AUTHOR);
- `/page/ROWSET/ROW[@num > 1 and @num < 4]`: sub-trees representing the second and third result rows of all queries (using the attribute num in the canonic output);
- `/page/ROWSET/ROW[last()]`: sub-trees representing the last row of all queries;
- `//ROW[contains(TITLE,'Galaxy')]`: all result rows where the TITLE field contains the string 'Galaxy';
- `count(//AUTHOR)`: the number of AUTHOR nodes;
- `name(/page/ROWSET[1]/ROW[1]/*)`: the name of the first column of the first query;
- `/page/ROWSET[1]/ROW | /page/ROWSET[2]/ROW`: the union of the nodes representing the result rows of the first and the second queries.

Quick reminder: if an XPath expression is used on an XSL stylesheet then due to the XML syntax reserved characters (`<`, `&`) must be replaced by the respective entity references.

7 Assigning XSL stylesheets to XML documents

Generally, generic XML documents do not carry information about how to display the data. Without using XSLT, a generic XML document is rendered as raw XML text by most web browsers. In order to style the rendering in a browser the XML document must include a reference to the stylesheet. This is done by defining the **xml-stylesheet** processing element in the source document. The obligatory value of attribute type is "text/xml", the stylesheet reference is provided in the attribute href:

```
<?xml-stylesheet type="text/xml" href="myStylesheet.xml"?>
```

When using XSQL templates stylesheets can be dynamically referenced by providing the value of attribute href by using a parameter. The following example illustrates how the minimum price for a book is displayed in XML or text format based on the value of the style form parameter. The XSQL template executing the query:

```
<?xml version="1.0" encoding="ISO-8859-2"?>
<?xml-stylesheet type="text/xml" href="{ @style }.xml"?>
<xsql:query connection="workshop" xmlns:xsql="urn:oracle-xsql">
  SELECT isbn, min(price) as price FROM book GROUP BY isbn
</xsql:query>
```

In the selected XML output every book is assigned a `<book>` tag, wherein ISBN is an attribute and where the value of the tag is the minimum price of that book:

```
<?xml version="1.0" encoding="ISO-8859-2"?>
```

```

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <pricelist>
    <xsl:for-each select="ROWSET/ROW">
      <book isbn="{ ISBN}"><xsl: value-of select="PRICE"/></book>
    </xsl:for-each>
  </pricelist>
</xsl:template>
</xsl:stylesheet>

```

As it can be inferred from the example the **value of an attribute** can be set on the stylesheet using the following action element: `<tag attribute="{XPath expression}">`.

In the text output there is a line for every book where ISBN and the minimum price are displayed:

```

<?xml version="1.0" encoding="ISO-8859-2"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="text"/>
<xsl:template match="/">
  <xsl:for-each select="ROWSET/ROW">
    <xsl: value-of select="ISBN"/> <xsl: value-of select="PRICE"/>
    <xsl: text>&#10;</xsl: text>
  </xsl:for-each>
</xsl:template>
</xsl:stylesheet>

```

The entity reference `
` in line 7 corresponds to the line-break character. Since the XSL transformation is eliminating the whitespace characters from the beginning and the end of text nodes in the stylesheet the `xsl: text` action element is used. The **xsl:text** action element sends the specified text to the output unchanged.

In XSQL templates the stylesheet set by the `<?xml-stylesheet?>` processing instruction can be overwritten using the special call parameter **xml-stylesheet**. Setting its value to “none” the XSL transformation can be easily turned off, for example:

<http://thereisno.net/minprice.xsql?xml-stylesheet=none>

8 Conditional tests and variables on XSL stylesheets

8.1 Conditional test

```

<xsl:if test="XPath expression">
  XSLT code... some output if the expression is true...
</xsl:if>

```

If the XPath expression provided in the test attribute is true then the embedded XSLT code segment is executed. If the XPath expression returns an XML node-set as result – and not a logical value – its value is true when the sub-tree is not empty. Thus the `<xsl:if>` element (and the `<xsl:choose>` element described below) is capable of answering is a given node exists inside the XML document, or not.

8.2 Multiple conditional test

```
<xsl:choose>
  <xsl:when test="XPath expression">
    XSLT code... some output if the expression is true...
  </xsl:when>
  <xsl:when test="XPath expression">
    XSLT code... some output if the expression is true...
  </xsl:when>
  ...
  <xsl:otherwise>
    XSLT code... some output if the expression is true...
  </xsl:otherwise>
</xsl:choose>
```

The `<xsl:choose>` element executes exactly one XSLT segment: the one where the XPath expression is first true. Otherwise the `xsl:otherwise` branch is executed.

8.3 Variables

The XSL variable name is deceiving, since XSL variables are constants and can be assigned a value only once. Variables are valid within the XML tag where the variable was defined and in all its descendants. Variables can be assigned scalar values (text, number, Boolean) or XML sub-trees as it is illustrated the following two definitions⁴:

```
<xsl:variable name="variable_name" select="XPath expression"/>
<xsl:variable name="variable _name">XML subtree</xsl:variable>
```

The defined variables can later be used in XPath expressions, where they can be referenced in the following form: `$variable_name`⁵. For example the:

```
<xsl:variable name="n" select="2"/>
<xsl:value-of select="item[$n]"/>
```

elements insert the 2nd item sub-tree in the output.

9 XSLT Templates

9.1 Recursive template processing

As it has been mentioned before, an XSLT template contains the transformation rules of a node. For every template there is a pattern specified using an XPath expression that defines on which nodes the template is matched. The pattern should be provided in the attribute `match`. In the previous examples only one template matching the root element was used, therefore it contained the transformation rules for the entire tree. The transformation can be performed in a recursive fashion using multiple templates.

⁴ Beware: in XSQL parameters the *value*; in XSL variable the *select* attribute is used to assign values!

⁵ In XSLT 1.0 the variables defined as XML result tree fragment are stored in text format and can only be handled as strings. Thus XPath node address expressions cannot be applied. However, result tree fragments can be inserted into the output as sub-trees using the `<xsl:copy-of>` element.

Generally the XSL transformation is performed, as follows. At the beginning of the XSL transformation the root node is the only selected node. The XSLT processor finds the template matching the selected node, and executes the transformation rules defined in the template. If the template contains an XSLT element selecting other nodes for further processing (typically child nodes) processing continues by selecting those other nodes one-by-one.

The XSLT processor contains a built-in template providing recursive processing in case there is not an explicit template on the stylesheet matching the element node:

```
<xsl:template match="*/">
  <xsl:apply-templates/>
</xsl:template>
```

This template does not provide any output; it only commands the processor to carry on processing the child nodes using the **xsl:apply-templates** element.

However, there is another built-in template for processing text and attribute nodes. This template simply copies the node value to the output:

```
<xsl:template match="text()|@">
  <xsl:value-of select="."/>
</xsl:template>
```

If there is more than one template matching a node, then the XSLT processor will always select the most specific one. As a rule of thumb, SOMETHING type patterns are more specific than the * pattern; but SOMETHING/SOMETHINGELSE or SOMETHING[condition] type patterns are even more specific.

The following example illustrates how a list of books can be displayed in HTML using recursive templates.

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="html"/>
  <xsl:template match="/">
    <html>
      <body><xsl:apply-templates/></body>
    </html>
  </xsl:template>

  <xsl:template match="ROWSET">
    <table><xsl:apply-templates/></table>
  </xsl:template>

  <xsl:template match="ROW">
    <tr><xsl:apply-templates/></tr>
  </xsl:template>

  <xsl:template match="ROW/*">
    <td><xsl:apply-templates/></td>
  </xsl:template>
</xsl:stylesheet>
```

9.2 Reusable and named templates

XSLT also enables organizing XSLT code segments used in multiple templates into separate blocks, also known as *named templates*. Unlike in case of ordinary templates, named templates are not selected by pattern matching. Named templates are called by the explicit **<xsl:call-template name="template_name">** element. The name of the named templates should be specified in the name attribute of the xsl:template element.

The following example illustrates how parameters are passed to a called template:

```
<?xml version="1.0" encoding="ISO-8859-2"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template name="named_template">
  <xsl:param name="par1" select="2"/>
  <!-- XSLT code -->
</xsl:template>

<xsl:template match="ROW">
  <xsl:call-template name="named_template">
    <xsl:with-param name="par1" select="42"/>
  </xsl:call-template>
</xsl:template>

</xsl:stylesheet>
```

9.3 Organizing stylesheets

To organize stylesheets more effectively, there is a way to segment s stylesheet into several files or to use template libraries stored in external stylesheets. The **xsl:include** and the **xsl:import** elements can be used to insert an external stylesheet – both elements should be specified as immediate child nodes of xsl:stylesheet. In case of xsl:include the precedence of the inserted templates is equal to; while in case of xsl:import is lower than that of the calling stylesheet templates.

10 Demo application

The demo application presented here demonstrates the search interface of a simple online bookshop. Searches can be made based on title, author or publisher. The XSQL template contains two queries: one returns the number of results while the other displays a result list. To avoid security flaws parameter SQL queries are used. On the user interface only 5 results are displayed at one time, the user can navigate between the results using the “Previous” and the “Next” links. By clicking on the author / publisher fields within a result list row a new list is generated displaying all books by that author / publisher. The most recent books are highlighted by having a “New” icon next to their row.

books.xsql

```
<?xml version="1.0" encoding="ISO-8859-2"?>
<?xml-stylesheet type="text/xsl" href="book_list.xsl"?>
<page connection="workshop"
  xmlns:xsql="urn:oracle-xsql"
  author="%"
```



```

title="%"
publisher="%"
search-key="none"
skip="0"
max-rows="5">

```

```

<!--Parameter values are passed on to the XSL-->
<xsql:include-param name="search-key"/>
<xsql:include-param name="search-value"/>
<xsql:include-param name="skip"/>
<xsql:include-param name="max-rows"/>

<!--Setting the search criteria parameter -->
<!--Perform this little magic in order to avoid -->
<!--security flaws caused by direct SQL manipulation -->
<xsql:set-page-param name="{@search-key}" value="{@search-value}%"/>

<!--Query the number of results -->
<xsql:set-page-param name="num-results"
    bind-params="author title publisher">
    SELECT COUNT(*) FROM book
    WHERE author LIKE ?
    AND title LIKE ?
    AND publisher LIKE ?
</xsql:set-page-param>
<xsql:include-param name="num-results"/>

<!--Query the results -->
<xsql:query rowset-element="STORE"
    row-element="BOOK"
    skip-rows="{@skip}"
    max-rows="{@max-rows}"
    bind-params="author title publisher"
    date-format="yyyy-MM-dd">
    SELECT * FROM book
    WHERE author LIKE ?
    AND title LIKE ?
    AND publisher LIKE ?
    ORDER BY author, title
</xsql:query>
</page>

```

book_list.xml

```

<?xml version="1.0" encoding="ISO-8859-2"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<!--Setting HTML output and code page -->
<xsl:output method="html"
    media-type="text/html"
    encoding="ISO-8859-2"/>

```

```

<xsl:variable name="base" select="'books.xsql'"/>

<!--Template matching the root element -->
<xsl:template match="/">
  <html>
    <head>
      <title>We recommend</title>
      <link rel="stylesheet" href="books.css" />
    </head>
    <body>
      <xsl:choose>
        <!--Error processing -->
        <xsl:when test="//xsql-error">
          <p>Execution error: <xsl:value-of select="//xsql-error/message[1]"/></p>
        </xsl:when>
        <xsl:otherwise>
          <h1>We recommend</h1>
          <xsl:call-template name="search_form"/>
          <xsl:call-template name="results_header"/>
          <xsl:apply-templates select="page/STORE"/>
        </xsl:otherwise>
      </xsl:choose>
    </body>
  </html>
</xsl:template>

<!--Displaying the search form -->
<xsl:template name="search_form">
  <p><form method="get">
    <b>Search: </b>
    <select name="search-key">
      <option value="title">
        <xsl:if test="/page/search-key/text()='title'">
          <xsl:attribute name="selected"/>
        </xsl:if>Title
      </option>
      <option value="author">
        <xsl:if test="/page/search-key/text()='author'">
          <xsl:attribute name="selected"/>
        </xsl:if>Author
      </option>
      <option value="publisher">
        <xsl:if test="/page/search-key/text()='publisher'">
          <xsl:attribute name="selected"/>
        </xsl:if>Publisher
      </option>
    </select>
    <input name="search-value" type="text" size="50" value="{ /page/search-value}"/>
  </form>
</p>
</xsl:template>

```

```

<!--Displaying results, Next and Previous fields -->
<xsl:template name="results_header">
<table width="100%" bgcolor="yellow">
<tr>
<td width="20%" align="left">
<xsl:if test="number(/page/skip) &gt;= number(/page/max-rows)">
<xsl:call-template name="results_header_href">
<xsl:with-param name="label" select="'Previous'"/>
<xsl:with-param name="skip" select="/page/skip - /page/max-rows"/>
</xsl:call-template>
</xsl:if>
</td>
<td align="center"><b>Results:
<xsl:choose>
<!--If there are any results, at all -->
<xsl:when test='/page/STORE/BOOK'>
<xsl:value-of select="page/skip + 1"/>
</xsl:when>
<xsl:otherwise>0</xsl:otherwise>
</xsl:choose> -
<xsl:choose>
<xsl:when test="number(/page/skip + /page/max-rows) &lt;= number(/page/num-
results)">
<xsl:value-of select="/page/skip + /page/max-rows"/>
</xsl:when>
<xsl:otherwise>
<xsl:value-of select="/page/num-results"/>
</xsl:otherwise>
</xsl:choose> /
<xsl:value-of select="page/num-results"/>
</b></td>
<td width="20%" align="right">
<xsl:if test="number(/page/skip + /page/max-rows) &lt;= number(/page/num-results)">
<xsl:call-template name="results_header_href">
<xsl:with-param name="label" select="'Next'"/>
<xsl:with-param name="skip" select="/page/skip + /page/max-rows"/>
</xsl:call-template>
</xsl:if>
</td>
</tr>
</table>
</xsl:template>

<!--Displaying label and link -->
<xsl:template name="results_header_href">
<xsl:param name="label" select="'Error'"/>
<xsl:param name="skip" select="0"/>

<xsl:choose>
<!--If there is no search criteria -->
<xsl:when test="/page/search-value = ''">
<a href="{ $base }?skip={ $skip }">
<xsl:value-of select="$label"/>

```

```

    </a>
</xsl:when>
<!--If there is a search criteria -->
<xsl:otherwise>
  <a href="{ $base}?search-key={ /page/search-key} & search-value={ /page/search-
value} & skip={ $skip}">
    <xsl:value-of select="$label"/>
  </a>
</xsl:otherwise>
</xsl:choose>
</xsl:template>

<!--Displaying search results -->
<xsl:template match="BOOK">
  <xsl:variable name="publishing_year" select="substring(PUBLISHED,1,4)"/>
  <p>
    <xsl:if test="number($publishing_year) > 2004">
      <xsl:text> </xsl:text>
    </xsl:if>
    <span class="title">
      <xsl:apply-templates select="AUTHOR"/> <xsl:apply-templates select="TITLE"/>
    </span>
    <br/>
    <span class="publisher">
      <b>Kiadó: <xsl:apply-templates select="PUBLISHER"/></b>
      (<xsl:value-of select="$publishing_year"/>)
    </span>
    <br/>
    <span>
      List price:
      <span class="listprice">
        <xsl:call-template name="print_price">
          <xsl:with-param name="amount" select="LIST_PRICE"/>
        </xsl:call-template>
      </span>
      Reduced price:
      <span class="price">
        <xsl:call-template name="print_price">
          <xsl:with-param name="amount" select="PRICE"/>
        </xsl:call-template>
      </span>
      You save:
      <span class="price">
        <xsl:call-template name="print_price">
          <xsl:with-param name="amount" select="LIST_PRICE - PRICE"/>
        </xsl:call-template>
        <xsl:variable name="save" select="100 * (LIST_PRICE - PRICE) div PRICE"/>
        (<xsl:value-of select="format-number($save,'0.0')"/>%)
      </span>
    </span>
  </p>
  <!--No separation line required after the last result -->
  <xsl:if test="position() &lt; last()">

```

```

        <hr/>
    </xsl:if>
</xsl:template>

<!-- Author: create a search link on the name -->
<xsl:template match="AUTHOR">
    <a href="{ $base}?search-key=author&search-value={.}">
        <xsl:apply-templates/>
    </a>
</xsl:template>

<!-- Title: detailed book information could be linked here -->
<xsl:template match="TITLE">
    <b><xsl:apply-templates/></b>
</xsl:template>

<!-- Publisher: create a search link on the name -->
<xsl:template match="PUBLISHER">
    <a href="{ $base}?search-key=publisher&search-value={.}">
        <xsl:apply-templates/>
    </a>
</xsl:template>

<!--Format price -->
<xsl:template name="print_price">
    <xsl:param name="amount" select="0"/>
    $<xsl:value-of select="format-number($amount,'0.00')"/>
</xsl:template>

</xsl:stylesheet>

```

The CSS stylesheet that belongs to the demo application can be viewed in the online version.

11 Getting ready

In order to successfully complete the workshop XSQL, XSLT and XPath specific knowledge is essential. It is assumed that participants are familiar with SQL and the also have basic HTML and UNIX skills.

1. If necessary, make a quick recap on previous SQL, HTML and UNIX studies. Read through the XSQL documentation.
2. Reviewing and understanding the demo application is key part of the preparation process. Download and test the demo application on the web server. The source can be downloaded from the workshop's website.
3. Modify the demo application – while using the common XSQL template – so that the results can be displayed in text format and in a HTML table format, as well.

12 References

T. Bray et al. (editors): *Extensible Markup Language (XML) 1.0 (Third Edition)*, W3C Recommendation, 2004
 J. Clark (editor): *XSL Transformations (XSLT) Version 1.0*, W3C Recommendation, 1999

J. Clark et al. (editors): *XML Path Language (XPath) Version 1.0*, W3C Recommendation, 1999
Oracle10g: *XML Developer's Kit Programmer's Guide*, Oracle Co., 2004
S. Muench: *Building Oracle XML Applications, Chapter 7: Transforming XML with XSLT*, O'Reilly, 2000

13 Appendix: XSQL Reference

xsql:dml	
Executes a SQL DML statement or PL/SQL anonymous block.	
commit	If its value is YES, a COMMIT is automatically triggered after the execution. The default value is NO.
bind-params	In case of using parameter SQL queries the XSQL parameters to be substituted are listed here. The parameters are listed in the order of usage, separated by a SPACE.
xsql:if-param	
This action allows you to conditionally include the elements and/or actions that are nested inside it if some condition is true. If the condition evaluates to true, then all nested XML content and XSQL actions are included in the page. This action was introduced in Oracle 10g.	
name	You specify which parameter value will be evaluated by supplying the required name attribute. Both simple parameter names as well as array-parameter names are supported.
exists	Tests whether the named parameter exists and has a non-empty value. The exists="yes" condition is true if the parameter exists and its value is not null. The exists="no" condition is true if the parameter does not exist or its value is null.
equals	This tests whether the named parameter equals the string value provided. The equals="stringValue" condition is true if the parameter value equals the stringValue.
not-equals	This tests whether the named parameter does not equal the string value provided. The not-equals="stringValue" condition is true if the parameter value does not equal the stringValue.
xsql:include-param	
This action enables you to include an XML representation of a single parameter in your output.	
name	The name of the parameter whose value you want to include.
xsql:include-request-params	
Include all request parameters as XML elements in your XSQL page.	
xsql:include-xml	
Include arbitrary XML resources at any point in your page by relative or absolute URL.	
href	Relative or absolute URL to an XML resource.
xsql:include-xsql	
Include the results of one XSQL page at any point inside another.	
href	Relative or absolute URL to an XSQL resource.

xsql:query	
Execute an arbitrary SQL statement and include its result in canonical XML format.	
bind-params	Ordered, space-delimited list of one or more XSQL parameter names whose values will be used to bind to the JDBC bind variable in the appropriate sequential position in the SQL statement.
date-format	Date format mask to use for formatted date column/attribute values in XML being queried. Valid values are those documented for the java.text.SimpleDateFormat class.
max-rows	Maximum number of rows to fetch, after optionally skipping the number of rows indicated by the skip-rows attribute. If not specified, default is to fetch all rows.
null-indicator	Indicates whether to signal that a column's value is NULL by including the NULL="Y" attribute on the element for the column. By default, columns with NULL values are omitted from the output. Valid values are yes and no. The default value is no.
row-element	XML element name to use instead of the default <ROW> element name for the entire row-set of query results.
rowset-element	XML element name to use instead of the default <ROWSET> element name for the entire row-set of query results.
skip-rows	Number of rows to skip before fetching rows from the result set. Can be combined with max-rows for stateless paging through query results.
xsql:set-cookie	
Set an HTTP Cookie.	
name	Name of the cookie whose value you want to set.
bind-params	Ordered, space-delimited list of one or more XSQL parameter names whose values will be used to bind to the JDBC bind variable in the appropriate sequential position in the SQL statement.
ignore-empty-value	Indicates whether the cookie assignment is ignored if the value to which it is being assigned is an empty string. Valid values are yes and no. The default value is no.
max-age	Sets the maximum age of the cookie in seconds. Default is to set the cookie to expire when the user terminates the current browser session.
only-if-unset	Indicates whether the cookie assignment only occurs when the cookie currently does not exist. Valid values are yes and no. The default value is no.
immediate	Indicates whether the cookie assignment is immediately visible to the current XSQL page. Typically cookies set in the current request are not visible until the browser sends them back to the server in a subsequent request. Valid values are yes and no. The default value is no.

xsql:set-page-param	
Set a page-level (local) parameter that can be referred to in subsequent SQL statements in the page.	
name	Name of the page-private parameter whose value you want to set.
bind-params	Ordered, space-delimited list of one or more XSQL parameter names whose values will be used to bind to the JDBC bind variable in the appropriate sequential position in the SQL statement.
ignore-empty-value	Indicates whether the page-level parameter assignment is ignored if the value to which it is being assigned is an empty string. Valid values are yes and no. The default value is no.
xsql:set-session-param	
Set an HTTP-Session level parameter.	
name	Name of the session-level variable whose value you want to set.
bind-params	Ordered, space-delimited list of one or more XSQL parameter names whose values will be used to bind to the JDBC bind variable in the appropriate sequential position in the SQL statement.
ignore-empty-value	Indicates whether the session-level parameter assignment is ignored if the value to which it is being assigned is an empty string. Valid values are yes and no. The default value is no.
only-if-unset	Indicates whether the session variable assignment only occurs when the session variable currently does not exist. Valid values are yes and no. The default value is no.
xsql:set-stylesheet-param	
Set the value of a top-level XSLT stylesheet parameter.	
name	Name of the top-level stylesheet parameter whose value you want to set.
bind-params	Ordered, space-delimited list of one or more XSQL parameter names whose values will be used to bind to the JDBC bind variable in the appropriate sequential position in the SQL statement.
ignore-empty-value	Indicates whether the stylesheet parameter assignment is to be ignored if the value to which it is being assigned is an empty string. Valid values are yes and no. The default value is no.

14 Appendix: XPath Function Reference

Node-set functions	
last()	Returns an integer equal to the context size from the expression evaluation context.
position()	Returns an integer equal to the context position from the expression evaluation context.
count(node-set)	Returns an integer representing the number of nodes in a node-set.
namespace-uri(node-set)	Returns a string representing URI of the namespace in which the given node resides.
name(node-set)	The name function returns a string representing the QName of the first node in a given node-set.
String manipulation functions	
string(object)	Converts the given argument to a string.
concat(string,string,...)	Concatenates two or more strings and returns the resulting string.
starts-with(string,string)	Checks whether the first string starts with the second string and returns true or false.
contains(string,string)	Determines whether the first argument string contains the second argument string and returns Boolean true or false.
substring-before(string,string)	Returns a string that is the rest of a given string before a given substring.
string-length(string?)	Returns a number equal to the number of characters in a given string. If the argument is omitted, string used will be the same as the context node converted to a string.
normalize-space(string?)	Eliminates leading and trailing white-space from a string, replaces sequences of whitespace characters by a single space, and returns the resulting string. If the argument is omitted, string used will be the same as the context node converted to a string.
translate(string ₁ ,string ₂ ,string ₃)	Evaluates a string and a set of characters to translate and returns the translated string. string ₁ is the string to evaluate. string ₂ are the string of characters that will be replaced. string ₃ are the string of characters used for replacement. The first character in string ₃ will replace every occurrence of the first character in string ₂ that appears in string ₁ . For example translate(„PHP“,„PH“,„DT“) results in „DTD“.
Logical functions	
boolean(object)	The Boolean function evaluates an expression and returns true or false. The expression can refer to numbers and node-sets as well as Booleans.
not(boolean)	The not function evaluates a Boolean expression and returns the opposite value.
true()	The true function returns a Boolean value of true.
false()	The false function returns Boolean false.

Numerical functions	
number(object)	Converts an object to a number and returns the number.
sum(node-set)	Returns a number that is the sum of the numeric values of each node in a given node-set.
floor(number)	Evaluates a decimal number and returns the largest integer less than or equal to the decimal number.
ceiling(number)	Evaluates a decimal number and returns the smallest integer greater than or equal to the decimal number.
round(number)	Returns a number that is the nearest integer to the given number.

15 Appendix: XSLT Reference

xsl:apply-templates	
In the absence of a select attribute, the xsl:apply-templates instruction processes the XSL transformation all of the children of the current node, including text nodes. Otherwise it processes the XSL transformation on the node-set defined by the select attribute.	
select	A select attribute can be used to process nodes selected by an XPath expression instead of processing all children. The value of the select attribute is an expression. The expression must evaluate to a node-set.
mode	If an xsl:apply-templates element has a mode attribute, then it applies only to those template rules from xsl:template elements that have a mode attribute with the same value; if an xsl:apply-templates element does not have a mode attribute, then it applies only to those template rules from xsl:template elements that do not have a mode attribute.
xsl:attribute	
The xsl:attribute element can be used to add attributes to result elements whether created by literal result elements in the stylesheet or by instructions such as xsl:element.	
name	The name attribute is interpreted as an attribute value template.
xsl:call-template	
An xsl:call-template element invokes a template by name; it has a required name attribute that identifies the template to be invoked. Parameters are passed to templates using the xsl:with-param element.	
name	The name of the template.
xsl:choose	
The xsl:choose element selects one among a number of possible alternatives. It consists of a sequence of xsl:when elements followed by an optional xsl:otherwise element.	
xsl:copy-of	
The xsl:copy-of element can be used to insert a result tree fragment into the result tree, without first converting it to a string as xsl:value-of does.	
select	The required select attribute contains an XPath expression that defines the node-set.

xsl:for-each	
The xsl:for-each instruction contains a template, which is instantiated for each node selected by the expression specified by the select attribute. The nodes are processed in document order, unless a sorting specification is present.	
select	The required select attribute contains an XPath expression that defines the node-set.
xsl:if	
The xsl:if element has a test attribute, which specifies an expression. The content is a template. The expression is evaluated and the resulting object is converted to a Boolean as if by a call to the Boolean function. If the result is true, then the content template is instantiated; otherwise, nothing is created.	
test	The XPath expression that is evaluated.
xsl:import	
The xsl:import element is only allowed as a top-level element. The xsl:import element children must precede all other element children of an xsl:stylesheet element, including any xsl:include element children. When xsl:include is used to include a stylesheet, any xsl:import elements in the included document are moved up in the including document to after any existing xsl:import elements in the including document.	
href	The URI reference identifying the stylesheet to be imported.
xsl:include	
The xsl:include element is only allowed as a top-level element. The inclusion works at the XML tree level. The resource located by the href attribute value is parsed as an XML document, and the children of the xsl:stylesheet element in this document replace the xsl:include element in the including document.	
href	The URI reference identifying the stylesheet to be included.
xsl:otherwise	
The xsl:choose element selects one among a number of possible alternatives. It consists of a sequence of xsl:when elements followed by an optional xsl:otherwise element. If no xsl:when is true, the content of the xsl:otherwise element is instantiated. If no xsl:when element is true, and no xsl:otherwise element is present, nothing is created.	
xsl:output	
The xsl:output element allows stylesheet authors to specify how they wish the result tree to be output. If an XSLT processor outputs the result tree, it should do so as specified by the xsl:output element.	
method	The overall method that should be used for outputting the result tree. The method must be one of xml, html or text.
encoding	Specifies the preferred character encoding that the XSLT processor should use to encode sequences of characters as sequences of bytes.
omit-xml-declaration	Specifies whether the XSLT processor should output an XML declaration; the value must be yes or no
indent	Specifies whether the XSLT processor may add additional whitespace when outputting the result tree; the value must be yes or no.
media-type	Specifies the media type (MIME content type) of the data that results from outputting the result tree.

xsl:param	
A variable is a name that may be bound to a value. The value to which a variable is bound (the value of the variable) can be an object of any of the types that can be returned by expressions. The value specified on the xsl:param variable is only a default value for the binding; when the template or stylesheet within which the xsl:param element occurs is invoked, parameters may be passed that are used in place of the default values.	
name	A required name attribute specifies the name of the variable.
select	If the variable-binding element has a select attribute, then the value of the attribute must be an XPath expression and the value of the variable is the object that results from evaluating the expression.
xsl:sort	
Sorting is specified by adding xsl:sort elements as children of an xsl:apply-templates or xsl:for-each element. The first xsl:sort child specifies the primary sort key, the second xsl:sort child specifies the secondary sort key and so on.	
select	An XPath expression that specifies the sort key.
data-type	Specifies the data type of the sort keys; the following values are allowed: text specifies that the sort keys should be sorted lexicographically; number specifies that the sort keys should be converted to numbers and then sorted according to the numeric value.
order	Specifies whether the strings should be sorted in ascending or descending order, the default is ascending.
case-order	Has the value upper-first or lower-first; this applies when data-type="text", and specifies that upper-case letters should sort before lower-case letters or vice-versa respectively.
xsl:stylesheet	
Root node of the stylesheet.	
version	An xsl:stylesheet element must have a version attribute, indicating the version of XSLT that the stylesheet requires. For this version of XSLT, the value should be 1.0.
xsl:template	
Defines a template.	
match	An XPath expression that identifies the source node or nodes to which the rule applies. The match attribute is required unless the xsl:template element has a name attribute.
name	An xsl:template element with a name attribute specifies a named template. If the template is not named, the name attribute might be omitted.
priority	The priority of a template rule is specified by the priority attribute on the template rule. The value of this must be a real number (positive or negative).
mode	If xsl:template does not have a match attribute, it must not have a mode attribute.

xsl:text	
Literal data characters may also be wrapped in an xsl:text element which copies them unchanged into the output.	
xsl:value-of	
The xsl:value-of element is instantiated to create a text node in the result tree.	
select	The required select attribute is an expression; this expression is evaluated and the resulting object is converted to a string as if by a call to the string function. The string specifies the string-value of the created text node. If the string is empty, no text node will be created. The created text node will be merged with any adjacent text nodes.
xsl:variable	
A variable is a name that may be bound to a value. The value to which a variable is bound (the value of the variable) can be an object of any of the types that can be returned by expressions.	
name	A required name attribute specifies the name of the variable.
select	If the variable-binding element has a select attribute, then the value of the attribute must be an XPath expression and the value of the variable is the object that results from evaluating the expression.
xsl:when	
The xsl:choose element selects one among a number of possible alternatives. It consists of a sequence of xsl:when elements followed by an optional xsl:otherwise element. Each xsl:when element has a single attribute, test, which specifies an expression. The content of the xsl:when and xsl:otherwise elements is a template.	
test	Each xsl:when element has a single attribute, test, which specifies an XPath expression
xsl:with-param	
Parameters are passed to templates using the xsl:with-param element .xsl:with-param is allowed within both xsl:call-template and xsl:apply-templates.	
name	The required name attribute specifies the name of the parameter.
select	If the variable-binding element has a select attribute, then the value of the attribute must be an XPath expression and the value of the variable is the object that results from evaluating the expression.