From XSLT to XQuery

This chapter presents XQuery to programmers with some experience with XSLT stylesheets. XQuery and XSLT have distinct goals but share several common features. Most of this "common ground" makes it easier to understand XQuery if you have XSLT programming experience.

To illustrate the similarity between the two languages we will take the XQuery use cases as an introduction to XQuery features, and then compare these constructs with equivalent or similar XSLT stylesheet declarations. We also will discuss the main differences between these two languages, making it a little easier to decide whether to apply XSLT or XQuery to solve a programming problem.

By the end of this chapter you will:

- Know the design goals for each language
- Distinguish the processing model used in XSLT and XQuery processors
- Understand the key conceptual differences between the languages
- Be able to compare XQuery built-in operators and their XSLT counterparts
- Be able to translate XSLT stylesheets to XQuery expressions

Presenting XQuery

Before showing language use cases and comparing operators and language processing models it's probably a good idea to take a 30,000 feet view of XSLT and XQuery, and examining their most significant features. Even when working on a day-to-day basis with XSLT, a programmer does not usually care about the design goals that directed the job of the language creators. This information now is useful in tracing a boundary between XQuery and XSLT.

We will mention fragments of the W3C specifications for both languages to help us figure what an XSLT user can expect from the new language.

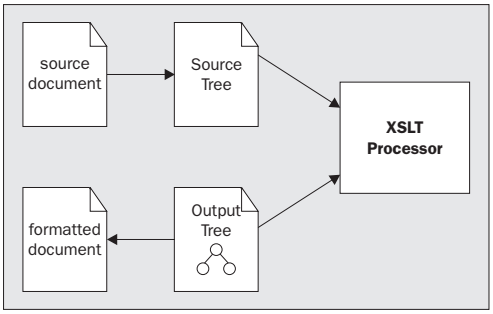
XSLT Design Goal

XSLT is a transformation language designed to work as a component of XSL – the "Extensible Stylesheet Language". This language's original goal was to provide a transformation tool able to convert a given XML document to a new document with suitable formatting information. The W3C XSLT Recommendation (http://www.w3.org/TR/xslt) helps describes XSLT with this note:

> *"XSLT may be used independently of XSL. However, XSLT is not intended as a completely general-purpose XML transformation language. Rather it is designed primarily for the kinds of transformations that are needed when XSLT is used as part of XSL."*

XSL architects divided the language in two parts: XSL-FO defines formatting semantics, and XSLT transforms XML data and generates adequate input to an XSL formatting processor. XSLT's design goal is to specifically meet requirements that express how XML-based structured content should be presented.

Figure 1



It is reasonable to say that an XSLT processor provides tools to transform a single XML document into a new formatted document. This can be useful for presentation, filtering, and several other purposes.

XQuery Design Goals

XQuery designers proposed quite different goals when specifying the requirements of the language. The XQuery 1.0 Working Draft states:

"[XQuery] is designed to be a small, easily implementable language in which queries are concise and easily understood. It is also flexible enough to query a broad spectrum of XML information sources, including both databases and documents."

XQuery was not meant to specifically handle document transformations. It is not even bound to operate with single documents; far from this, XQuery is required to be as flexible as possible when handling heterogeneous data sources. The W3C *XML Query Requirements* document (http://www.w3.org/TR/xmlquery-req) lists several uses; some of them are part of XSLT, and several are not adequate for an XSLT engine:

- Perform queries on human-readable documents
- Process mixed-model documents (with structured and human-readable contents)
- Process data-oriented documents (structured, machine-readable content)
- Be embedded in multiple programming environments
- Operate directly on DOM structures
- Process administrative data (log files, etc.)
- Perform catalogue searches
- Do stream filtering (like UNIX command-line tools)
- Be a front-end to native XML databases and web servers

Every time we talk about document translation, we see that XQuery can do tasks that XSLT is supposed to handle well. In addition, all tasks related to large, stream-like, or distributed data are new and uncovered ground. These are the main reasons the W3C proposed a new query language.

There are comments on the W3C XQuery mailing list presenting some key differences between XQuery and XSLT. We can enumerate ease-of-use (simple queries look simpler in XQuery than in XSLT), similarity with conventional database commands (there are several samples of this in previous chapters), optimizing abilities (XQuery resembles other database-related technologies providing an algebra in the formal semantic language definition), and strong typing (the final language recommendation will provide type enforcement rules similar to those favored in production environments).

Comparing XQuery to XSLT

The XQuery requirements document states that the language must be able to handle document transformations and composition; this is almost a rewriting of the XSLT role definition. However, we can acquire a little more information on the common features of both languages.

Common Ground

What can XSLT do for us? The language is supposed to:

- Be able to process XML mark-up with bare data and generate a presentation-ready document
- Apply transformations according to pattern matches
- Use XPath to identify portions of the source document tree
- Walk a source document tree both programmatically and using template rules (predicate logic)

"XQuery has better mechanisms to work with structured and distributed XML data."

XQuery transformation capabilities suppose the language should:

- Be able to query multiple XML data sources to compose a result in the XQuery/XPath data model
- Apply transformations according to pattern-matches
- Use XPath to identify portions of the source document tree
- Walk several source document trees using template matching

The common ground is how both languages match source data and trigger transformations and element constructors. XPath was picked as the preferred matching tool, allowing both languages to easily transverse source XML trees.

A Look at the Differences

The large difference between the two languages can easily be stated as:

XQuery has better mechanisms to work with structured and distributed XML data.

This is the single biggest factor in favor of XQuery over XSLT for a transformation task. XSLT provides more refined transformation mechanisms, while XQuery allows very simple expressions to query structured data. Do your needs include the processing of one document at a time? Both languages can handle this. However, if you want to query several *unrelated* documents to compose a single answer, or to process a continuous data stream with a filter, XQuery is the easiest choice. Of course, there are other XML query languages, but XQuery is the only one actually backed by a W3C Working Draft.

We can examine more details on how the two languages differ. A few things you can do with XSLT that XQuery will not easily provide are:

- XSLT template rules let you describe easily what changes in a document (XQuery expressions should specify everything that goes in the output – it can copy a whole element tree but you need to declare it using recursive functions; elegant for simple tasks but it may become cumbersome for complex transformations)
- Create variables visible by the whole stylesheet (XQuery variables are **bound** to an expression result and can be referenced only in subsequent clauses of the same expression)
- Generate HTML output (XQuery only generate results on the XQuery/XPath data model – sequences of documents or fragments of well-formed XML)
- Recursively apply several templates in a source document, letting the engine decide which template takes precedence to handle the transformation (to do recursive evaluation with XQuery you need to do it explicitly, defining a recursive function and calling it from your query)

All of the above makes XQuery a lot more suited to handle data with strict and repeated formatting, in contrast with XSLT's ability to easily describe minor updates in a document. Let us now enumerate what XQuery can provide with its transformation capabilities that you would not be able to reproduce with XSLT:

- Apply path expressions on more than one source tree (different documents) and easily compare or apply set operations on the results (XSLT deals with more than one document at a time (each has to be specified by name) but makes it difficult to deal with all of them at once)
- Define functions to be used in path expressions (XSLT offers callable templates but nothing that can be used within an XPath expression)
- Match text across element boundaries (XSLT only provides tools to compare contents of single elements and attributes)
- Perform set operations on the result of path operations (XSLT applies templates on each element that matches a pattern, but it does not filter a node sequence by comparing it to other sequences)
- Express subsets of items of a path (XSLT 1.1 uses XPath 1.0, which lets you refer only to sibling items in a path)

The transformation features in XQuery are best employed in sentences to extract fragments of a document; the language does not make it easy to transform a whole document, to add markup, or perform global restructure.

The W3C *XQuery Use Cases* document (http://www.w3.org/TR/xmlquery-use-cases) provides several samples involving document transformations, but all of them return fragments of the original documents, never the full data structure. The usual context where XQuery is effective is when you wish to analyze the logical representation of a data-source and present a summary of it.

Where You Should Expect to See XQuery

A developer may use XQuery processing engine to handle transformation tasks, but it frequently will be simpler to handle these by using XQuery only to retrieve data and to make suitable arrangements of documents as adequate input to an XSLT engine.

If you want to combine two documents, it can be done with a little pre-processing. A filter may read both documents and compose a single one placing the root element of each document as children of a new parent root element. (We will present a sample of this later on this chapter.) However, what if you have two huge documents and need only small fragments of them? You would get the best of both worlds by using XQuery to extract relevant data from the huge documents and generate a small "query answer", and then applying an XSLT stylesheet to present it properly. If you prefer, you can use XQuery to transform documents in this way in place of XSLT.

We will learn XQuery tools to build output and see how easy it is to create a well-formed XML document as a query answer. Then we will learn what XQuery can do for us in matters not related to formatting at all.

"XQuery is expected to be a superset of XPath 2.0."

XPath Expressions: XQuery Flavor

One of XQuery's requirements was the use of XPath syntax to identify source data. This is a consistent approach, which coincides with the XSL working group objective to provide a tool to navigate the hierarchical structure of an XML document, and for it to be used for matching (testing whether or not a node matches a pattern).

XPath 1.0 was designed to handle XSLT and XPointer requirements. XQuery however added some new requisites, so the premises for a new XPath specification (2.0) were defined. While there is not a final version of XPath 2.0, XQuery designers specified an XPath extension described in the XQuery Working Draft. The final XPath 2.0 Recommendation should be backwards-compatible with XPath 1.0, but will add features to fit requirements for XQuery 1.0 and for XSLT 2.0. The XQuery 1.0 Recommendation mentions, "XQuery is expected to be a superset of XPath 2.0". XSLT 2.0 will share this specification as its expression language.

The Data Model

The biggest change in XPath 2.0 is the data model it works on. The language is still built around the same path axes and the same step/predicate structure, but the data-source, and the results of a path expression, now belong to a larger domain. The new data model offers the following improvements:

- Support for all XML Schema data types. The XPath used in XQuery supports both the data structures and simple data types proposed by the XML Schema working group.
- The path expression may operate on and return both "collections of documents", and values of schema data types. XPath 1.0 was restricted to a node-set, a number, a Boolean, or a string.

Node-Sequences, Not Node-Sets

The new data model dropped the node-set concept. The data model defines the scope where a path expression operates as "*a document, a fragment of well-formed XML, a primitive value or a list whose elements can be any of the former*". This defines the concept of **node-sequence**, a list of node-hierarchies where each element is well-formed XML (not necessarily a full document; may be a sequence of simple data types, documents, and document fragments).

The node-sequence enforces an order for its items, and the XQuery processor will be able to generate a response that keeps this order. The order for a sequence may or may not follow the original document order. Several XQuery constructs let you control the ordering in a sequence.

Node-sequences made the old node-set concept obsolete – the newer construct is more general and also handles the XPath 1.0 use cases. Sequences may also contain duplicate items, unlike the node-sets where an item always may appear only once. XQuery also does not use node-sets because they do not provide mechanisms to represent ordered query results.

XPath is a Key Feature of XQuery

XQuery is a functional language where every sentence is an expression and every expression returns values belonging to the language data model. This arrangement was devised to make it easy to formulate complex queries where each operates on the result of sub-expressions. Any XQuery query is an expression and an XPath expression is a valid XQuery expression. So the simplest case of XQuery is a single path expression, totally valid as input to the query engine. The following is a perfectly valid query:

```
/foo/bar
```

This will return all of the `bar` children of the `foo` root element as the query result. Let us examine some sample path expressions formulated in the extended XQuery syntax. We will formulate the queries on the following sample document, `census.xml`, extracted from the *XQuery Use Cases* document.

The census data describes two families that have several intermarriages.

```
<?xml version="1.0"?>
<!DOCTYPE census [
  <!ELEMENT census (person*)>
  <!ELEMENT person (person*)>
  <!ATTLIST person
        name    ID      #REQUIRED
        spouse  IDREF   #IMPLIED
        job     CDATA   #IMPLIED >
]>
<!-- census.xml -->
<census>
  <person name="Bill" job="Teacher">
    <person name="Joe" job="Painter" spouse="Martha">
      <person name="Sam" job="Nurse">
        <person name="Fred" job="Senator" spouse="Jane">
        </person>
      </person>
      <person name="Karen" job="Doctor" spouse="Steve">
      </person>
    </person>
    <person name="Mary" job="Pilot">
      <person name="Susan" job="Pilot" spouse="Dave">
      </person>
    </person>
  </person>
  <person name="Frank" job="Writer">
    <person name="Martha" job="Programmer" spouse="Joe">
      <person name="Dave" job="Athlete" spouse="Susan">
      </person>
    </person>
    <person name="John" job="Artist">
      <person name="Helen" job="Athlete">
      </person>
      <person name="Steve" job="Accountant" spouse="Karen">
        <person name="Jane" job="Doctor" spouse="Fred">
        </person>
      </person>
    </person>
  </person>
</census>
```

A simple query to find the job of a person called "Jane" written in XSLT would be:

```
<?xml version="1.0"?>
<!-- JaneQuery.xsl -->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
   <xsl:template match="//person[@name='Jane']">
      <xsl:value-of select="@job"/>
   </xsl:template>
</xsl:stylesheet>
```

Note: for readers who are not familiar with how to apply the samples shown, here is a suggestion. Java users may download the Apache Xalan XSLT processor, unpack its files and use its command line format as in

```
> java org.apache.xalan.xslt.Process -IN census.xml -XSL JaneQuery.xsl
  -OUT janejob.xml
```

Microsoft Internet Explorer users may rely on its internal XSLT engine by placing a special processing instruction inside the source XML tree and opening the file in the browser window. The processing instruction looks like:

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

Now look below at how this one-line query may be written in XQuery:

```
document("census.xml")//person[@name="Jane"]/@job
```

When testing this, we found that QuiP did not like the DOCTYPE declaration at the start of `census.xml`. Just remove this declaration if you have the same problems.

In this sample, XQuery uses XPath just like XSLT does. In fact there is nothing special about path expressions in XQuery. The big change from XSLT to XQuery is that XQuery avoids verbosity, making sentences simpler. As mentioned before, even an isolated XPath expression is a valid XQuery expression.

The XQuery version of XPath delivers the same information with less coding. We can extract all the doctors and senators from the data source with the samples below (first the XSLT version, then the XQuery one):

```
<?xml version="1.0"?>
<!-- DoctorsAndSenators.xsl -->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
  <xsl:template match="//person[@job='Doctor']">
    <xsl:copy-of select="."/>
  </xsl:template>
  <xsl:template match="//person[@job='Senator']">
    <xsl:copy-of select="."/>
```

```
  </xsl:template>
</xsl:stylesheet>
```

```
document("census.xml")//person[(@job="Doctor") or (@job="Senator")]
```

Both queries above will deliver the information shown overleaf as result:

```
<person name="Karen" job="Doctor" spouse="Steve" />
<person name="Fred" job="Senator" spouse="Jane" />
<person name="Jane" job="Doctor" spouse="Fred" />
```

The returned elements from XQuery come by default in the XPath defined **document order**, the sequence in which the element starting tags appear in the document serialized XML markup.

So performing queries with XQuery path expressions is much more concise than doing it with an XSLT stylesheet. There is no template handling, or XML stylesheet structure to worry about, just an expression rendering its answer directly. XQuery always returns the complete content of all items in the expression result; the items can be XML fragments or simple W3C Schema data types.

Positional Selection in Ranges

Instead of pointing to a simple element in a predicate, XPath 2.0 allows us to specify a list of elements by position, either by a list of literal numbers (as in `1, 3, 4, 7`) or by an expression that generates a sequence (as in `2 TO 5`). As in XPath 1.0 the first node in the sequence is considered to have the ordinal number 1.

```
document("zoo.xml")//chapter[2 TO 5]//figure
```

This finds all the figures in chapters 2 through 5 of the document named `zoo.xml`.

This is Q2 from "XQuery 1.0: An XML Query Language"

Dereference Operator

XQuery added to XPath the dereference operator (`=>`). This operator takes a node of type `IDREF` and returns the elements referenced by this value. The operator is always followed by an element test; if the test fails the element is not returned:

```
document("zoo.xml")//chapter[title = "Frogs"]//figref/@refid=>fig/caption
```

This finds captions of figures that are referenced by `figref` elements in the chapter of `zoo.xml` with title "`Frogs`".

This is Q3 from "XQuery 1.0: An XML Query Language"

Steps Made of XQuery Expressions

In XQuery any path step can be formed with XQuery expressions, extending the functionality of the path expression significantly. This addition to the XPath spec is yet to be corroborated by the joint XPath 2.0 task force of the XSL and XML Query working groups. The expression should be enclosed in parentheses, `(` and `)`, to avoid ambiguity. Here there is a sample union of two paths using this feature (the resulting node-sequence includes values where the step matches "`figure`" or "`table`"):

```
document("zoo.xml")//chapter[title = "Monkeys"]//(figure |
table)/caption
```

This finds all captions of figures and tables in the chapter of `zoo.xml` with the title "`Monkeys`".

Q5 from "XQuery 1.0: An XML Query Language"

Transformations in XQuery

XQuery provides convenient ways to compose new output, allowing any expression in the language to embed literal XML markup. To do this, just type well-formed XML data in the query text as you are used to doing inside `xsl:template` elements. This feature in XQuery is known as **element constructors** and it will handle almost all needs involving output of new elements, including element trees, CDATA sections, attributes, and so on. The only thing the specification still does not allow is the insertion of element references in queries, but this may be addressed before the next draft.

Every XQuery sentence is a valid "expression" in the language. Remember that an expression always returns valid constructs for the language data model, and a fragment of literal mark-up of course is a valid expression. Let us demonstrate the simplest transformation expressed both in XQuery and XSLT.

```
<?xml version="1.0"?>
<!-- Comment.xsl -->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
  <xsl:template match="/">
    <xsl:comment> output a comment and a single empty element
</xsl:comment>
    <sample/>
  </xsl:template>
</xsl:stylesheet>
```

The equivalent XQuery construct is:

```
<!--output a comment and a single empty element -->
<sample />
```

The above XQuery does not currently work in QuiP, as QuiP does not like XML comments.

The engine output for both is the same, but for a single item XQuery will not return an XML declaration at the beginning of the output, because the result of an XQuery expression is not necessarily a single XML document. The data model allows sequences of XML fragments and also simple data types.

> XQuery will not return an XML declaration at the beginning of the output, because the result of an XQuery expression is not necessarily and XML document.

The particular case shown is a well-formed XML fragment and some engine implementations may offer you tools to serialize it as a proper XML document. The samples above have no utility as real transformations, but they show the philosophy adopted by each language to provide transformations. XSLT is bound to the document structure, and even to emit just some literal markup you need to specify, "match the root element and then output this fragment". XSLT is very consistent in the way it handles document processing.

XQuery is meant to be concise. The sample just says, "Do not care about the data-source contents, just return this literally." XQuery's approach is geared toward the easy formulation and combination of simple queries, but it also provides some complex constructs that made it a little bit closer to XSLT.

Combining Path Expression with Element Constructors

Element constrsuctors by themselves do not transform anything. But when combined with XQuery path expressions they provide an extremely concise way to express simple transformations. Using the same sample document, `census.xml`, we can formulate a simple transformation to count the `person` element.

```
<?xml version="1.0"?>
<!-- countPeople.xsl -->
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <count-persons>
      <xsl:value-of select="count(//person)" />
    </count-persons>
  </xsl:template>
</xsl:stylesheet>
```

```
<count-persons>
  {count(document("census.xml")//person)}
</count-persons>
```

The above query returns the following:

```
<count-persons>14</count-persons>
```

XQuery uses a convention similar to what we found in XSLT: a pair of `{}` curly braces identify an expression to be evaluated. XSLT allows this syntax to be used in the value of certain attributes, while XQuery let you use it inside any literal XML mark-up.

In the above example, we see another big difference between XSLT and XQuery transformations: there is no such thing as a default template in XQuery. The only output you get is what you explicitly specified. XQuery provides no automatic mechanism to apply transformations, like XSLT's generic `<xsl:apply-templates />` command.

More Element Construction with FLWR Expressions

The simple element constructing technique shown above is only suitable for very simple XQuery transformations because the sentence formulated will not iterate over a collection of nodes, but it does show how easy is to display aggregate data with XQuery.

As seen in the previous chapter, XQuery provides a simple but yet powerful language construct to handle collection transversal, the FLWR (`FOR-LET-WHERE-RETURN`) expressions. FLWR expressions provide a construction mechanism very similar to what is provided in XSLT by the `xsl:template` and the `xsl:for-each` commands.

The general form for FLWR expressions is:

```
FOR variable IN expression
LET variable := expression
WHERE expression
RETURN expression
```

A FLWR expression may have several independent `FOR` and `LET` clauses. Each one defines a variable to be tested in the optional `WHERE` clause and to be used to compose output in the `RETURN` clause.

The `FOR` clause behaves very much like `xsl:for-each`. In XSLT this command accepts an XPath expression and iterates over it, placing each element as the context node. XQuery does a similar job, evaluating the given expression and binding each of its items to the declared variable.

```
FOR $myvar IN document("census.xml")//person/@job
RETURN $myvar
```

This returns the following:

```
Teacher
Writer
Painter
Pilot
```

```
Nurse
Doctor
Senator
Pilot
Programmer
Artist
Athlete
Athlete
Accountant
Doctor
```

The RETURN clause is responsible for result construction. It comprises a single expression where any of the previously defined variables can be used. Each time it is called, RETURN will construct a piece of the expression result; here we see how XQuery adapts well to handle multiple repetitions of small constructs.

Be careful to not take the "result construct" role as "query output". RETURN constructs the FLWR expression output, and this will be the query output if the FLWR expression is the topmost expression in the query. XQuery lets you build a query with nested expressions, and the RESULT clause of inner expressions will compose the expression output but not necessarily the query results.

The following query demonstrates this. It uses a FLWR expression to find all the "Doctors" in the census.xml document, and then an outer path extracts just the first element from the FLWR result:

```
(FOR $pers IN document("census.xml")//person[@job="Doctor"]/@name
 RETURN
   $pers)[1]
```

This returns:

```
Karen
```

The LET clause is more like the xsl:variable directive. It will store any valid data in the XQuery data model – it can be a sequence of simple data types or nodes, created with a path expression. The bound variable will then be subject to tests in the WHERE clause and result composition in the RETURN clause, exactly like the variables defined in the FOR clause. The previous query could be rewritten with LET to deliver the same result.

```
LET $perslist := document("census.xml")//person[@job="Doctor"]/@name
RETURN
   $perslist[1]
```

Note that now the path expression to extract the first element may be the inner expression, because in the RESULT clause the $perslist variable assumes the value of the sequence of name attributes, not the individual value for each one.

The WHERE clause is a supercharged template matching mechanism. It works as the RETURN clause: it is called for each combination of distinct values of the bound variables created with FOR/LET to define a match and provides transformation tools comparable to the XSLT xsl:if, xsl:choose, xsl:when, and xsl:otherwise commands. The sample below shows how to list the name for all the doctors with a FLWR expression.

```
FOR $myvar IN document("census.xml")//person
WHERE $myvar/@job = "Doctor"
RETURN $myvar/@name
```

This result is:

```
Karen Jane
```

Another powerful feature of the XQuery WHERE clause is its ability to handle results from path expressions applied in several different data sources at once. It is really simple; just prepare suitable bound variables in the former FOR and LET clauses. This lets you join segregated data based on their content – a concept very common for database developers.

Let us formulate a transformation to show the power of more complex FLWR expressions. We will output a flat list of all person elements whose profession is "Athlete", deprived of their content if any. First we examine a XSLT stylesheet for comparison purposes, and then the same transform will be shown as an XQuery FLWR expression.

```
<?xml version="1.0"?>
<!-- findAthlete.xsl -->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
  <xsl:template match="/">
    <persons>
      <xsl:for-each select="//person[@job='Athlete']">
        <person>
          <xsl:copy-of select="@*"/>
        </person>
      </xsl:for-each>
    </persons>
  </xsl:template>
</xsl:stylesheet>
```

The equivalent XQuery is:

```
<persons>
  {
    FOR $p IN document("census.xml")//person
    WHERE $p/@job='Athlete'
    RETURN
      <person>
        {$p/@*}
```

```
      </person>
  }
</persons>
```

Both should return the following XML:

```
<persons>
  <person name="Dave" job="Athlete" spouse="Susan" />
  <person name="Helen" job="Athlete" />
</persons>
```

XQuery FLWR expressions are very convenient for handling database applications, which usually require extraction or summarizing from data sources. The FLWR expression format makes it easy to express the handling of long repetitions of similar elements.

The example above flattens a hierarchical structure but it is very similar to an expression to iterate over a list-like collection. Most XQuery expressions deal with the kind of construct where you iterate over a list and produce some construction with it. Note that XQuery is very concise and clear once you know the meaning of each construct.

> "XQuery is very concise and clear once you know the meaning of each construct."

More on Expression Chaining

A single XQuery FLWR expression is already a very powerful mechanism for transformations. We can boost its power by chaining expressions. This is XQuery's way to deliver greater result composability in queries. The base concept is that expressions take other expressions as parameters – the query result is built upon successive transforms, where an expression result is passed to another expression, which uses it to compose another result, and so on.

XQuery enforces that any expression in the language should return results in the language data model, and each clause in a FLWR expression may accept a nested FLWR expression. The W3C *XQuery Use Cases* shows a good example of expression chaining to handle complex transformations. The following examples work on the `bib.Xml` document, first shown in Chapter 4 and repeated below:

```
<!-- bib.xml -->
<bib>
  <book year="1994">
    <title>TCP/IP Illustrated</title>
    <author><last>Stevens</last><first>W.</first></author>
    <publisher>Addison-Wesley</publisher>
    <price> 65.95</price>
  </book>
  <book year="1992">
    <title>Advanced Programming in the Unix environment</title>
    <author><last>Stevens</last><first>W.</first></author>
    <publisher>Addison-Wesley</publisher>
```

```
    <price>65.95</price>
  </book>
  <book year="2000">
    <title>Data on the Web</title>
    <author><last>Abiteboul</last><first>Serge</first></author>
    <author><last>Buneman</last><first>Peter</first></author>
    <author><last>Suciu</last><first>Dan</first></author>
    <publisher>Morgan Kaufmann Publishers</publisher>
    <price> 39.95</price>
  </book>
  <book year="1999">
    <title>The Economics of Technology and Content for Digital
TV</title>
    <editor>
       <last>Gerbarg</last><first>Darcy</first>
       <affiliation>CITI</affiliation>
    </editor>
    <publisher>Kluwer Academic Publishers</publisher>
    <price>129.95</price>
  </book>
</bib>
```

The following sample transformation inverts the document structure, listing each author and the list of book titles he published. First we show the XQuery, then the corresponding XSLT version.

```
<results>
  {
    FOR $a IN distinct(document("bib.xml")//author)
    RETURN
      <result>
        {$a }
        {
          FOR $b IN document("bib.xml")//book[author = $a]
          RETURN $b/title
        }
      </result>
  }
</results>
```

```
<?xml version="1.0"?>
<!-- listByAuthor.xsl -->
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <results>
      <xsl:for-each select="//author[not(.=preceding::author)]">
```

```
        </results>
      </xsl:template>

<xsl:template match="//author">
        <result>
<xsl:copy of select="."/>
<xsl:for each select=".">
</xsl:for each>
</result>
```

The example query above uses the `distinct()` function from XQuery's core library. It operates on a node-sequence and ensures no individual node value will be repeated in the result.

The program fragment calling the `distinct()` function was copied from the *XQuery Use Cases* document. This function should have been defined in complementary XQuery documentation, and it was redefined on August 27 in the *XQuery 1.0 and XPath 2.0 Functions and Operators Version 1.0* W3C Working Draft.

The function is now called `value-distinct()` to differentiate it from the `identity-distinct()` operator, who ensures no node will be repeated but based on node-identity and not node value. Check the results:

```
<results>
  <result>
    <author>
      <last>Stevens</last>
      <first>W.</first>
    </author>
    <title>TCP/IP Illustrated</title>
    <title>Advanced Programming in the Unix environment</title>
  </result>
  <result>
    <author>
      <last>Abiteboul</last>s
      <first>Serge</first>
    </author>
    <title>Data on the Web</title>
  </result>
  <result>
    <author>
      <last>Buneman</last>
      <first>Peter</first>
    </author>
    <title>Data on the Web</title>
  </result>
  <result>
    <author>
```

```
      <last>Suciu</last>
      <first>Dan</first>
    </author>
    <title>Data on the Web</title>
  </result>
</results>
```

Expressions in the `WHERE` and `RETURN` clauses of a FLWR expression will inherit all the variables defined in the `FOR` and `LET` clauses of the top expression – the variables in scope. In the previous example, the inner FLWR does a path predicate test using a variable from the outer expression.

This is similar to `xsl:call-template` in the way the outer expression prepares bound variables with values and makes them in visible in the scope of the inner expression. This behavior also applies to the list of `FOR` and `LET` clauses in a single expression: each variable declared will be visible in the expressions for the subsequent clauses.

Looking at some of the previous examples, we may think XQuery is just a very different approach to solve the same transformation problems. This is only partially true, because here we are comparing both languages' constructs to explain XQuery features. XSLT 1.0 offers 36 different elements for stylesheet composition, while XQuery has only XPath expressions, element constructors, FLWR expressions, and some special conditional mechanisms. Certainly there are transformations that easily done with XSLT and nightmarish to do with XQuery.

The previous XQuery sample shows an unusual situation for XSLT, an expression calling another and later generating results. The XSLT output mechanisms operate immediately and directly in the resulting document. XQuery expressions will not generate any output by themselves. In XQuery, the query engine output is defined by the results of the topmost expression. Any inner expression, as complex as it is, will only deliver results to its caller; it is up to the caller expression to reproduce these values to the output or to just use them for further computations. XQuery expressions will not generate any output by themselves until the topmost expression returns its result.

Comparing Transformation Capabilities

In all of the former use cases, XQuery features were introduced and compared to XSLT commands. Later, we looked at a few code samples. Most XQuery sentences were very clear and concise, appearing to be more directly to the point. This perspective may tempt us to wrongly judge each language's ability to handle document transformations.

In reality XQuery is not a very good candidate to substitute XSLT as a document transformation tool. From the previous examples, the reader can infer a significant XQuery feature: it is easy to describe a transformation when you mention every tree fragment that composes the result tree. XSLT behavior on this issue is much more coherent for the transformation role, because template rules allow you to easily describe only the changes in the document.

You can preserve data to be left untouched in an XQuery transform, but you need to add code telling the query engine to do this task. XSLT by default modifies only specific portions of a document and leaves the rest of it unchanged. XQuery can do it, with recursive functions, but XQuery does not make it the easiest task.

Now we will see a use case where XQuery does compare well with XSLT. We will make a subtle change in the `bib.xml` sample document, switching the `price` elements with `price` attributes on the `book` element.

Here we have a possible XSLT solution for this task:

```
<?xml version="1.0"?>
<!-- priceSwitch.xsl -->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
  <xsl:template match="//book">
    <book>
      <xsl:attribute name="price">
        <xsl:value-of select="price" />
      </xsl:attribute>
      <xsl:copy-of select="@*" />
      <xsl:for-each select="*[name()!='price']">
        <xsl:copy-of select="."/>
      </xsl:for-each>
    </book>
  </xsl:template>
</xsl:stylesheet>
```

Note that the stylesheet describes only the transformation, not mentioning unrelated elements in the document. Now we will formulate an XQuery able to do the same transformation on the sample document:

```
<bib>
{
  FOR $b IN document("bib.xml")/bib/book
  LET $p  := $b/price
  RETURN
    <book price={$p}>
      {$b/@year}
      {
        FOR $child IN $b/*
        WHERE name($child) != "price"
        RETURN $child
      }
    </book>
}
</bib>
```

This results in:

```
<bib>
  <book year="1994" price="65.95">
    <title>TCP/IP Illustrated</title>
    <author>
      <last>Stevens</last>
      <first>W.</first>
    </author>
    <publisher>Addison-Wesley</publisher>
  </book>
  <book year="1992" price="65.95">
    <title>Advanced Programming in the Unix environment</title>
    <author>
      <last>Stevens</last>
      <first>W.</first>
    </author>
    <publisher>Addison-Wesley</publisher>
  </book>
  <book year="2000" price="39.95">
    <title>Data on the Web</title>
    <author>
      <last>Abiteboul</last>
      <first>Serge</first>
    </author>
    <author>
      <last>Buneman</last>
      <first>Peter</first>
    </author>
    <author>
      <last>Suciu</last>
      <first>Dan</first>
    </author>
    <publisher>Morgan Kaufmann Publishers</publisher>
  </book>
  <book year="1999" price="129.95">
    <title>
     The Economics of Technology and Content for Digital TV
    </title>
    <editor>
      <last>Gerbarg</last>
      <first>Darcy</first>
      <affiliation>CITI</affiliation>
    </editor>
    <publisher>Kluwer Academic Publishers</publisher>
  </book>
</bib>
```

The XQuery expression is not too complex, but this sample handles the transformation in a cumbersome way. The `RETURN` clause composes a new content for the `book` element reading the sequence of child nodes. To filter it, we compare each child element's name with `price` (name of the element to be purged). The problem with this query is that it will generate output comprising only the root element `bib`, its `book` child elements, and each `book's` child.

Another document dependency we find in this sample is the linear structure of the response. This FLWR expression will return a flat list of books, and its XSLT cousin is able to transform any tree whose `book` elements have a single `price` child element. To do the same with XQuery we should write a recursive function thus becoming able to keep the tree structure in the query result.

XQuery Imitates XSLT

In this part, we will present XQuery constructs that closely resemble their XSLT counterparts. The XQuery 1.0 Working Draft mentions several technologies that inspired the language design, but XSLT is not cited. This is probably because XSLT did not create the features shown below, but instead borrowed them from other languages.

Sorting Data

XQuery provides a sorting operator that can be applied to the result of any expression. The `SORTBY` clause follows the following general form:

```
expression SORTBY (expression direction, expression direction, ...)
```

The data to be sorted is the result of the expression on the left of the `SORTBY` clause. The list of expressions between the parentheses is the sort condition evaluations. Each individual value from the data to be sorted will be evaluated by these expression, and the individual results should support the > operator to enable the sorting engine to handle the task.

The engine uses the resulting values to compute a new, sorted, sequence for the list. To build the resulting list, the value for `direction` can be optionally declared, and can be `ASCENDING` (the default value) or `DESCENDING`.

Unlike XSLT's `xsl:sort` instruction, which only operates inside `xsl:apply-templates` and `xsl:for-each` elements, XQuery's `SORTBY` can be applied to any expression result, including of course FLWR expressions. To demonstrate, below is a sample query that produces a sorted list of `person` elements from the `census.xml` document:

```
<persons>
{
  FOR $p IN document("census.xml")//person
  RETURN
    <person>
```

```
      {$p/@*}
    </person>
  SORTBY (@name)
}
</persons>
```

The value `@name` in the sort expression list tells the engine to evaluate it as a relative XPath expression on each node generated by the FLWR expression to generate the output shown below:

```
<persons>
  <person name="Bill" job="Teacher">
  <person name="Dave" job="Athlete" spouse="Susan"/>
  <person name="Fred" job="Senator" spouse="Jane"/>
  <person name="Helen" job="Athlete"/>
  <person name="Jane" job="Doctor" spouse="Fred"/>
  <person name="Joe" job="Painter" spouse="Martha"/>
  <person name="Karen" job="Doctor" spouse="Steve"/>
  <person name="Martha" job="Programmer" spouse="Joe"/>
  <person name="Mary" job="Pilot"/>
  <person name="John" job="Artist"/>
  <person name="Sam" job="Nurse"/>
  <person name="Steve" job="Accountant" spouse="Karen"/>
  <person name="Susan" job="Pilot" spouse="Dave"/>
</persons>
```

The fragment of XSLT to do the same is very similar:

```
<?xml version="1.0"?>
<!-- sortCensus.xsl -->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
  <xsl:template match="/">
    <persons>
      <xsl:for-each select="//person">
        <xsl:sort select="@name" />
        <person>
          <xsl:copy-of select="@*"/>
        </person>
      </xsl:for-each>
    </persons>
  </xsl:template>
</xsl:stylesheet>
```

Defining Functions

This mechanism provided by XQuery to extend the language at first looks a lot like the XSLT `xsl:call-template` instruction. However, XQuery designers had goals that were more ambitious for XQuery functions. The requisite for XQuery functions includes:

- Define a function with a formal parameter list and a specific return data type (this basic capability supersedes that of `xsl:call-template`, because XSLT's callable templates can only set variables or generate output. Also, XQuery function parameters are strongly typed, whereas `xsl:template` parameters are not.
- Provide a set of rules to automatically convert instances of the XML Schema simple data types.
- Permit the definition of recursive and mutually recursive functions (a pair of functions where one calls the other and vice-versa).
- Optionally omit type declaration for parameters or return value, adopting standard types: parameters assume the type "any node" and return values the value node-sequence.
- Automatically apply a function designed to receive an item over a list of items of the same data type, and apply a function designed to receive a list to a single individual item (taking it as a one-item list).

A sample query where a function is used to determine the list of a `person` child in the sample `census.xml` document. The core XQuery `shallow()` function is used to return a copy of all the child elements for the element with the `name` attribute of "Joe" or for the element whose `name` is the equal to Joe's `spouse` attribute. It will return a flat list of elements with their attributes but without any of its children. The examples below show this, first with XSLT and than in XQuery.

```
<?xml version="1.0"?>
<!-- functionEquiv.xsl -->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
  <xsl:template match="/">
    <joechildren>
      <xsl:for-each select="//person[@name='Joe']/person">
        <xsl:call-template name="print-person">
          <xsl:with-param name="myperson" select="."/>
        </xsl:call-template>
      </xsl:for-each>
      <xsl:for-each
        select="//person[@name=(//person[@name='Joe']/@spouse)]/person">
        <xsl:call-template name="print-person">
```

```
          <xsl:with-param name="myperson" select="."/>
        </xsl:call-template>
      </xsl:for-each>
    </joechildren>
  </xsl:template>

  <xsl:template name="print-person">
    <xsl:param name="myperson"/>
    <person>
      <xsl:copy-of select="$myperson/@*"/>
    </person>
  </xsl:template>
</xsl:stylesheet>
DEFINE FUNCTION children (ELEMENT $p) RETURNS ELEMENT
{
  shallow($p/person) UNION shallow($p/@spouse=>person/person)
}
<result>
{
  FOR $j IN document("census.xml")//person[@name = "Joe"]
  RETURN children($j)
}
</result>
```


This did not work under QuiP at time of editorial, as QuiP did not support the `shallow()` *function.*

The samples above produce:

```
<joechildren>
  <person name="Sam" job="Nurse" />
  <person name="Karen" job="Doctor" spouse="Steve" />
  <person name="Dave" job="Athlete" spouse="Susan" />
</joechildren>
```

Note that the callable template does not set a return value, but directly adds content to the result tree. Some XSLT programmers use variables to simulate some of XQuery function behavior, but the resulting code is nowhere as simple or reusable as XQuery's `DEFINE FUNCTION` construct.

Recursive Functions

Recursive evaluation is the core of XSLT technology. All the `xsl:template` clauses are searched and in a match are recursively evaluated for each `xsl:apply-templates` command in the stylesheet.

A recursive function call is a convenient way to traverse a tree. It generates output while keeping the document hierarchy intact. XQuery functions allow us to mimic this XSLT feature and free ourselves from the flat return list model proposed by FLWR expressions.

To demonstrate this useful construct we will start from the `census.xml` document and make a new document with the same hierarchy but replacing the `job` and `spouse` attributes with equivalent child elements. Each node should be converted from:

```
<person name="Dave" job="Athlete" spouse="Susan" />
```

to the following form:

```
<person name="Dave">
  <job>Athlete</job>
  <spouse>Susan</spouse>
</person>
```

Let us first review the XSLT solution for this. As already stated, it is very concise due to the language's inherent ability to recursively evaluate transformations.

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
  <xsl:template match="/census/person">
    <census>
      <xsl:apply-templates select="person" />
    </census>
  </xsl:template>

  <xsl:template match="person">
    <xsl:copy>
      <xsl:apply-templates select="@name" />
      <job> <xsl:value-of select="@job" /> </job>
      <xsl:if test="count(@spouse) > 0">
        <spouse> <xsl:value-of select="@spouse"/> </spouse>
      </xsl:if>
      <xsl:apply-templates select="person" />
    </xsl:copy>
  </xsl:template>

  <xsl:template match="//person/@name">
    <xsl:copy-of select="."/>
  </xsl:template>
</xsl:stylesheet>
```

The recursive call is shown in gray. It is handled automatically by the XSLT engine because the "`person`" template expression matches the given `xsl:apply-templates` call. This renders the following output:

```
<census>
  <person name="Joe">
    <job>Painter</job>
    <spouse>Martha</spouse>
    <person name="Sam">
      <job>Nurse</job>
      <person name="Fred">
        <job>Senator</job>
        <spouse>Jane</spouse>
      </person>
    </person>
   ...
  </person>
  ...
</census>
```

Now we will write the same using an XQuery function:

```
DEFINE FUNCTION rewrite_person (ELEMENT $p) RETURNS ELEMENT
{
  <person>
    {$p/@name}
    <job>
      {$p/@job/text()}
    </job>
    {
      IF (count($p/@spouse) > 0)
      THEN <spouse>{$p/@spouse/text()}</spouse>
      ELSE ()
    }
    {
      FOR $child IN $p/person
      RETURN rewrite_person($child)
    }
  </person>
}

<census>
{
  FOR $aperson IN document("census.xml")/census/person
  RETURN rewrite_person($aperson)
}
</census>
```

Unlike XSLT, XQuery's approach to recursion requires the programmer to previously select values for formal parameters and then explicitly call the function with each of them. XSLT also allows you to do it with callable templates, but the engine default behavior is the automatic evaluation of all the suitable templates found in the stylesheet.

The XQuery expression above also shows us the language's conditional operator, in the format:

```
IF expression1 THEN expression2 ELSE expression3
```

If the first expression evaluates to `true` the `THEN` clause is evaluated and its results returned; otherwise, the expression in the `ELSE` clause receives the same treatment.

XQuery Set/Sequence-Oriented Constructs

XQuery modified the original XPath data model to provide better support handling portions of a data source. This design was reflected initially in the range positional predicates – the `[x TO y]`, or `[k, l, m]` formats. The language also provides several operators able to combine or modify sequences of values. An XQuery sequence does not need to include only nodes from the document tree. Any simple data type can compose a sequence and most of the language's operators will be able to operate on them as well.

Sequence Operations

The basic sequence operator is the comma `","`; it is placed between two expressions and combines them into an ordered sequence of values.

```
1, 2
```

The sample above is a valid XQuery expression that generates a sequence with two simple data elements.

The `TO` operation presented in path expressions at the beginning of this chapter is also a sequence constructor. It is like the `","` (comma) operator and converts both parameters to integers before constructing a sequence of values including all the values from the left to the right operands. So the expression:

```
1 TO 4
```

means exactly the same as:

```
1, 2, 3, 4
```

The sequential character of XQuery data may be used in the language's conditional expressions through the `BEFORE` and `AFTER` clauses. `BEFORE` is an infix operator that takes two sequences as parameters and returns nodes from the first sequence that are at least before one node of the second sequence. This is possible only if the two sequences are subsets of the same larger sequence. `AFTER` works in a similar way, taking elements from the first sequence that follow elements from the second.

```
document("bib.xml")//book BEFORE
  document("bib.xml")//book[2]
```

is equivalent to:

```
document("bib.xml")//book[1]
```

and will return the following one element sequence:

```
<book year="1994">
  <title>TCP/IP Illustrated</title>
  <author>
    <last>Stevens</last>
    <first>W.</first>
  </author>
  <publisher>Addison-Wesley</publisher>
  <price>65.95</price>
</book>
```

This can be very useful when used in path expressions. We can demonstrate it by extracting all books written after 1999 from the sample `bib.xml` document.

```
<results>
{
  LET $booklist := distinct(document("bib.xml")//book) SORTBY (@year)
  RETURN
    $booklist AFTER ($booklist/book[@year = "1999"])
}
</results>
```

```
<results>
  <book year="2000">
    <title>Data on the Web</title>
    <author><last>Abiteboul</last><first>Serge</first></author>
    <author><last>Buneman</last><first>Peter</first></author>
    <author><last>Suciu</last><first>Dan</first></author>
    <publisher>Morgan Kaufmann Publishers</publisher>
    <price>39.95</price>
  </book>
</results>
```

In the previous example the book list is first sorted and stored in a variable, so we are able to refer to it twice in the AFTER sequence operation. If the data were not sorted previously, the expression would return an empty book list because the last book element in the document has the year attribute equal to 1999.

This shows clearly the difference between document order and sequence order. XQuery expressions always work in sequence order. Specifically with node-sequences this may be the standard XPath document order – if the sequence did not change due to any sort or sequence operation.

Set Operations

XQuery may also operate on node-sequences disregarding element order, as if they were element sets. The language provides three set operators for this task:

- UNION (or "|")
- INTERSECT
- EXCEPT

They all take two sequences as arguments and generate a single result sequence. UNION has an alternate syntax, "|" to resemble XPath 1.0 syntax. The following expression will return the list of all books written in 1992 or published by "Addison-Wesley":

```
document("bib.xml")//book[@year="1992"] UNION
  document("bib.xml")//book[publisher/text()="Addison-Wesley"]
```

The first expression results in a single book, and the second returns two. The union of both is:

```
<book year="1994">
  <title>TCP/IP Illustrated</title>
  <author><last>Stevens</last><first>W.</first></author>
  <publisher>Addison-Wesley</publisher>
  <price> 65.95</price>
</book>
<book year="1992">
  <title>Advanced Programming in the Unix environment</title>
  <author><last>Stevens</last><first>W.</first></author>
  <publisher>Addison-Wesley</publisher>
  <price>65.95</price>
</book>
```

XQuery returns only two book nodes because the UNION operator will not repeat a node. This is evaluated with the node identity concept, as is the identity-distinct() function. This is also the behavior of node-sequences in the XPath 1.0 Data Model.

The INTERSECT operator provides the same functionality as set intersection in mathematics. It will only return nodes present in both sequences, so the expression:

```
document("bib.xml")//book[@year="1992"] INTERSECT
  document("bib.xml")//book/publisher[text()="Addison-Wesley"]
```

returns a single book node that appears in both parameters for the INTERSECT operator.

```
<book year="1992">
  <title>Advanced Programming in the Unix environment</title>
  <author><last>Stevens</last><first>W.</first></author>
  <publisher>Addison-Wesley</publisher>
  <price>65.95</price>
</book>
```

EXCEPT performs as the mathematical set difference operator. It also takes two sequences and returns only elements present in the first but not in the second sequence.

```
document("bib.xml")//book/publisher[text()="Addison-Wesley"] EXCEPT
  document("bib.xml")//book[@year="1992"]
```

This returns the following XML:

```
<book year="1994">
  <title>TCP/IP Illustrated</title>
  <author><last>Stevens</last><first>W.</first></author>
  <publisher>Addison-Wesley</publisher>
  <price>65.95</price>
</book>
```

Quantifiers (SOME / EVERY)

XQuery simplifies writing expression predicates iterating over any sequence to verify if each or some individual items satisfy a condition. The SOME and EVERY expressions always return a Boolean value (also a valid XQuery expression) where:

```
EVERY variable IN expression1 SATISFIES expression2
```

This is true only if all items in the sequence returned by *expression1* evaluate to true when applied to *expression2*, such as:

```
EVERY $n IN 1, 2, 3 SATISFIES $n > 0
```

The SOME construct has the same general form but will evaluate to true if any of the values satisfies the condition:

```
SOME $n in 1 TO 3 SATISFIES $n = 2
```

Both constructs above are useful to apply logical `AND` and `OR` operators through a whole set of values. Be careful to note:

EVERY will always return true if the expression to be iterated over is empty

XQuery Influence in XSLT 2.0

XSLT 2.0 evolved from previous specifications to accommodate the language role as a formatting tool, not a query mechanism. The W3C *XSLT Requirements Version 2.0* working draft document states this clearly in its opening section:

"XSLT 2.0 has the following goals:

- Simplify manipulation of XML Schema-typed content
- Simplify manipulation of string content
- Support related XML standards
- Improve ease of use
- Improve interoperability
- Improve i18n support
- Maintain backward compatibility
- Enable improved processor efficiency

In addition, the following are explicitly not goals:

- Simplifying the ability to parse unstructured information to produce structured results
- Turning XSLT into a general-purpose programming language"

The XSLT working group is acting in sync with the XQuery working group to define clear roles for each language. XSLT will evolve to become a better transformation tool for presentation purposes, while keeping compatibility with XSLT 1.0. The proposal also states that XSLT was not meant to be a general-purpose programming language nor to be specifically suited to generate structured results.

XQuery gave significant contributions to the ongoing XPath 2.0 specification. XSLT 2.0 will inherit all this, so both languages will share a common ground in path expression syntax. However, new XQuery constructs will be geared toward operating on XML data-sources with the addition of commands to update XML data and to define persistent views of XML databases.

The development of XQuery and related XML technologies and the feedback of the XSLT user community gave XSLT 2.0 designers some hints to improve the language handling of several tasks:

- Data grouping
- XML Schema data types (mostly handled by the XPath 2.0 core function library)
- The direct use of `IDREF` attributes when selecting paths (also a feature built into XPath 2.0.)

The arrangement of tools provided today points to developers using XSLT mostly for presentation purposes. Programmers will rely on XQuery to access database systems and extract data to feed to the XSL formatter.

Summary

This chapter showed us some of the strengths and weakness of XQuery when compared to the current XSLT recommendation. It also presented key XQuery features to handle XML transformations, like:

- The XPath 2.0 Data Model
- The sequence concept
- Ranges in path expressions
- Transforms with constructors and FLWR expressions
- Expression chaining
- Data sorting
- Defining recursive and non-recursive functions
- Set, sequence, and quantifier operators

In the following chapters we will look at these concepts in use in early XQuery implementations.

early adopter

XQuery