Part I

BASICS

Chapter 1

XQUERY: A GUIDED TOUR

Jonathan Robie

XML **(Extensible Markup Language)** is an extremely versatile data format that has been used to represent many different kinds of data, including web pages, web messages, books, business and accounting data, XML representations of **relational database** tables, programming interfaces, objects, financial transactions, chess games, vector graphics, multimedia presentations, credit applications, system logs, and textual variants in ancient Greek manuscripts.

In addition, some systems offer XML views of non-XML data sources such as relational databases, allowing XML-based processing of data that is not physically represented as XML. An XML document can represent almost anything, and users of an XML **query language** expect it to perform useful queries on whatever they have stored in XML. Examples illustrating the variety of XML documents and queries that operate on them appear in [XQ-UC].

However complex the data stored in XML may be, the structure of XML itself is simple. An XML document is essentially an outline in which order and hierarchy are the two main structural units. XQuery is based on the structure of XML and leverages this structure to provide query capabilities for the same range of data that XML stores. To be more precise, XQuery is defined in terms of the XQuery 1.0 and **XPath** 2.0 Data Model [XQ-DM], which represents the parsed structure of an XML document as an ordered, labeled tree in which **nodes** have **identity** and may be associated with simple or **complex types.** XQuery can be used to query XML data that has no **schema** at all, or that is governed by a **World Wide Web Consortium** (W3C) **XML Schema** or by a **Document Type Definition** (DTD). Note that the **data model** used by XQuery is quite different from the classical relational model, which has no hierarchy, treats order as insignificant, and does not support identity. XQuery is a **functional language**—instead of executing commands as procedural languages do, every query is an expression to be evaluated, and expressions can be combined quite flexibly with other expressions to create new expressions.

This chapter gives a high-level introduction to the XQuery language by presenting a series of examples, each of which illustrates an important feature of the language and shows how it is used in practice. Some of the examples are drawn from [XQ-UC]. We cover most of the language features of XQuery, but also focus on teaching the idioms used to solve specific kinds of problems with XQuery. We start with a discussion of the structure of XML documents as input and output to queries and then present basic operations on XML—locating nodes in XML structures using **path expressions,** constructing XML structures with **element constructors,** and combining and restructuring information from XML documents using **FLWOR expressions,** sorting, conditional expressions, and quantified expressions. After that, we explore operators and functions, discussing arithmetic operators, comparisons, some of the common functions in the XQuery function library, and how to write and call user-defined functions. Finally, we discuss how to import and use XML Schema **types** in queries.

Many users will learn best if they have access to a working implementation of XQuery. Several good implementations can be downloaded for free from the Internet; a list of these appears on the W3C XML Query Working Group home page, which is found at `http://www.w3.org/xml/Query.html`.

This chapter is based on the May 2003 Working Draft of the XQuery language. XQuery is still under development, and some aspects of the language discussed in this chapter may change.

Katz_C01.qxd 7/23/03 2:45 PM Page 5

Sample Data: A Bibliography

This chapter uses bibliography data to illustrate the basic features of XQuery. The data used is taken from the XML Query Use Cases, Use Case “XMP,” and originally appeared in [EXEMPLARS]. We have modified the data slightly to illustrate some of the points to be made. The data used appears in Listing 1.1.

Listing 1.1 Bibliography Data for Use Case “XMP”

```
<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>65.95</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 data for this example was created using a DTD, which specifies that a bibliography is a sequence of books, each book has a title, publication year (as an **attribute**), an author or an editor, a publisher, and a price, and each author or editor has a first and a last name, and an editor has an affiliation. Listing 1.2 provides the DTD for our example.

Listing 1.2 DTD for the Bibliography Data

```
<!ELEMENT bib  (book* )>
<!ELEMENT book  (title,  (author+ | editor+ ), publisher, price )>
<!ATTLIST book  year CDATA  #REQUIRED >
<!ELEMENT author  (last, first )>
<!ELEMENT editor  (last, first, affiliation )>
<!ELEMENT title  (#PCDATA )>
<!ELEMENT last  (#PCDATA )>
<!ELEMENT first  (#PCDATA )>
<!ELEMENT affiliation  (#PCDATA )>
<!ELEMENT publisher  (#PCDATA )>
<!ELEMENT price  (#PCDATA )>
```

Data Model

XQuery is defined in terms of a formal data model, not in terms of XML text. Every input to a query is an instance of the data model, and the output of every query is an instance of the data model. In the XQuery data model, every document is represented as a tree of nodes. The kinds of nodes that may occur are: document, **element,** attribute, text, **namespace,** processing instruction, and comment. Every node has a unique node identity that distinguishes it from other nodes—even from other nodes that are otherwise identical.

In addition to nodes, the data model allows **atomic values,** which are single values that correspond to the **simple types** defined in the W3C Recommendation, "XML Schema, Part 2" [SCHEMA], such as strings, Booleans, decimals, integers, floats and doubles, and dates. These simple types may occur in any document associated with a W3C XML Schema. As we will see later, we can also represent several simple types directly as literals in the XQuery language, including strings, integers, doubles, and decimals.

An **item** is a single node or atomic value. A series of items is known as a sequence. In XQuery, every value is a sequence, and there is no distinction between a single item and a sequence of length one. Sequences can only contain nodes or atomic values; they cannot contain other sequences.

The first node in any document is the document node, which contains the entire document. The document node does not correspond to anything visible in the document; it represents the document itself. Element nodes, comment nodes, and processing instruction nodes occur in the order in which they are found in the XML (after expansion of entities). Element nodes occur before their children—the element nodes, text nodes, comment nodes, and processing instructions they contain. Attributes are not considered children of an element, but they have a defined position in **document order:** They occur after the element in which they are found, before the children of the element. The relative order of attribute nodes is implementation-dependent. In document order, each node occurs precisely once, so sorting nodes in document order removes duplicates.

An easy way to understand document order is to look at the text of an XML document and mark the first character of each element **start tag,** attribute name, processing instruction, comment, or text node. If the first character of one node occurs before the first character of another node, it will precede that node in document order. Let's explore this using the following small XML document:

```
<!— document order —>
<book year="1994">
    <title>TCP/IP Illustrated</title>
    <author><last>Stevens</last><first>W.</first></author>
</book>
```

The first node of any document is the document node. After that, we can identify the sequence of nodes by looking at the sequence of start characters found in the original document—these are identified by underlines in the example. The second node is the comment, followed by the book element, the year attribute, the title element, the text node containing `TCP/IP Illustrated`, the author element, the last element, the text node containing `Stevens`, the first element, and the text node containing `W.`

Literals and Comments

XQuery uses “smiley faces” to begin and end comments. This cheerful notation was originally suggested by Jeni Tennison. Here is an example of a comment:

```
(: Thanks, Jeni! :)
```

Note that XQuery comments are comments found in a query. XML documents may also have comments, like the comment found in an earlier example:

```
<!— document order —>
```

XQuery comments do not create XML comments—XQuery has a constructor for this purpose, which is discussed later in the section on constructors.

XQuery supports three kinds of numeric literals. Any number may begin with an optional + or – sign. A number that has only digits is an integer, a number containing only digits and a single decimal point is a decimal, and any valid floating-point literal containing an `e` or `E` is a double. These correspond to the XML Schema simple types `xs:integer`, `xs:decimal`, and `xs:double`.

```
1       (: An integer :)
-2      (: An integer :)
+2      (: An integer :)
1.23    (: A decimal  :)
-1.23   (: A decimal  :)
1.2e5   (: A double   :)
-1.2E5  (: A double   :)
```

String literals are delimited by quotation marks or apostrophes. If a string is delimited by quotation marks, it may contain apostrophes; if a string is delimited by apostrophes, it may contain quotation marks:

```
"a string"
'a string'
"This is a string, isn't it?"
'This is a "string"'
```

If the literal is delimited by apostrophes, two adjacent apostrophes within the literal are interpreted as a single apostrophe. Similarly, if the literal is delimited by quotation marks, two adjacent quotation marks within the literal are interpreted as one quotation mark. The following two string literals are identical:

```
"a "" or a ' delimits a string literal"
'a " or a '' delimits a string literal'
```

A string literal may contain predefined entity references. The entity references shown in Table 1.1 are predefined in XQuery.

Here is a string literal that contains two predefined entity references:

```
'&lt;bold&gt;A sample element.&lt;/bold&gt;'
```

Input Functions

XQuery uses input functions to identify the data to be queried. There are two input functions:

1. `doc()` returns an entire document, identifying the document by a **Universal Resource Identifier** (URI). To be more precise, it returns the document node.
2. `collection()` returns a **collection**, which is any sequence of nodes that is associated with a URI. This is often used to identify a database to be used in a query.

TABLE 1.1 Entity References Predefined in XQuery

Entity Reference	Character Represented
`&lt;`	`<`
`&gt;`	`>`
`&amp;`	`&`
`&quot;`	`"`
`&apos;`	`'`

If our sample data is in a file named `books.xml`, then the following query returns the entire document:

```
doc("books.xml")
```

A **dynamic error** is raised if the `doc()` function is not able to locate the specified document or the `collection()` function is not able to locate the specified collection.

Locating Nodes: Path Expressions

In XQuery, path expressions are used to locate nodes in XML data. XQuery's path expressions are derived from XPath 1.0 and are identical to the path expressions of XPath 2.0. The functionality of path expressions is closely related to the underlying data model. We start with a few examples that convey the intuition behind path expressions, then define how they operate in terms of the data model.

The most commonly used operators in path expressions locate nodes by identifying their location in the hierarchy of the tree. A path expression consists of a series of one or more **steps,** separated by a slash, `/`, or double slash, `//`. Every step evaluates to a sequence of nodes. For instance, consider the following expression:

```
doc("books.xml")/bib/book
```

This expression opens `books.xml` using the `doc()` function and returns its document node, uses `/bib` to select the `bib` element at the top of the document, and uses `/book` to select the `book` elements within the bib element. This path expression contains three steps. The same books could have been found by the following query, which uses the double slash, `//`, to select all of the `book` elements contained in the document, regardless of the level at which they are found:

```
doc("books.xml")//book
```

Predicates are `Boolean` conditions that select a subset of the nodes computed by a step expression. XQuery uses square brackets around predicates. For instance, the following query returns only authors for which `last="Stevens"` is true:

```
doc("books.xml")/bib/book/author[last="Stevens"]
```

If a predicate contains a single numeric value, it is treated like a subscript. For instance, the following expression returns the first author of each book:

```
doc("books.xml")/bib/book/author[1]
```

Note that the expression `author[1]` will be evaluated for each book. If you want the first author in the entire document, you can use parentheses to force the desired precedence:

```
(doc("books.xml")/bib/book/author)[1]
```

Now let's explore how path expressions are evaluated in terms of the data model. The steps in a path expression are evaluated from left to right. The first step identifies a sequence of nodes using an input function, a variable that has been bound to a sequence of nodes, or a function that returns a sequence of nodes. Some XQuery implementations also allow a path expression to start with a `/` or `//`.

Such paths start with the root node of a document, but how this node is identified is implementation-defined. For each `/` in a path expression, XQuery evaluates the expression on the left-hand side and returns the resulting nodes in document order; if the result contains anything that is not a node, a **type error** is raised. After that, XQuery evaluates the expression on the right-hand side of the `/` once for each left-hand node, merging the results to produce a sequence of nodes in document order; if the result contains anything that is not a node, a type error is raised. When the right-hand expression is evaluated, the left-hand node for which it is being evaluated is known as the context node.

The step expressions that may occur on the right-hand side of a `/` are the following:

- A *NameTest*, which selects element or attribute nodes based on their names. A simple string is interpreted as an element name; we have already seen the *NameTest* `bib`, which evaluates to the `bib` elements that are children of the context node. If the name is prefixed by the @ character (pronounced "at"), then the *NameTest* evaluates to the attributes of the context node that have the specified name. For instance, `doc("books.xml")/bib/book/@year` returns the year attribute of each book. *NameTest* supports both namespaces and wildcards, which are discussed later in this section.

- A *KindTest*, which selects processing instructions, comments, text nodes, or any node based on the type of the node. The *KindTest* used to select a given kind of node looks like a function with the same name as the type of the node: `processing-instruction()`, `comment()`, `text()`, and `node()`.
- An expression that uses an explicit "axis" together with a *NameTest* or *KindTest* to choose nodes with a specific structural relationship to the context node. If the *NameTest* `book` selects `book` elements, then `child::book` selects book elements that are children of the context node; `descendant::book` selects book elements that are descendants of the context node; `attribute::book` selects book attributes of the context node; `self::book` selects the context node if it is a book element, `descendant-or-self::book` selects the context node or any of its descendants if they are book elements, and `parent::book` selects the parent of the context node if it is a book element. Explicit axes are not frequently used in XQuery.
- A *PrimaryExpression*, which may be a literal, a function call, a variable name, or a parenthetical expression. These are discussed in the next section of this tutorial.

Now let's apply what we have learned to the following expression:

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

Working from left to right, XQuery first evaluates the input function, `doc("books.xml")`, returning the document node, which becomes the context node for evaluating the expression on the right side of the first slash. This right-hand expression is `bib`, a *NameTest* that returns all elements named `bib` that are children of the context node. There is only one `bib` element, and it becomes the context node for evaluating the expression `book`, which first selects all `book` elements that are children of the context node and then filters them to return only the first book element.

Up to now, we have not defined the `//` operator in terms of the data model. The formal definition of this operator is somewhat complex; intuitively, the `//` operator is used to give access to all attributes and all descendants of the nodes in the left-hand expression, in document order. The expression `doc("books.xml")//bib` matches the `bib` element at the root of our sample document, `doc("books.xml")//book` matches all the `book` elements in the document, and `doc("books.xml")//@year` matches all the `year` attributes in the document. The `//` is formally defined using full axis notation: `//` is equivalent to `/descendant-or-self::node()/`.

For each node from the left-hand expression, the `//` operator takes the node itself, each attribute node, and each descendant node as a context node, then evaluates the right-hand expression. For instance, consider the following expression:

```
doc("books.xml")/bib//author[1]
```

The first step returns the document node, the second step returns the `bib` element, the third step—which is not visible in the original query—evaluates `descendant-or-self::node()` to return the `bib` element and all nodes descended from it, and the fourth step selects the first `author` element for each context node from the third step. Since only `book` elements contain `author` elements, this means that the first author of each book will be returned.

In the examples we have shown so far, *NameTest* uses simple strings to represent names. *NameTest* also supports namespaces, which distinguish names from different vocabularies. Suppose we modify our sample data so that it represents titles with the `title` element from the Dublin Core, a standard set of elements for bibliographical data [DC]. The namespace URI for the Dublin Core is http://purl.org/dc/elements/1.1/. Here is an XML document containing one simple book, in which the `title` element is taken from Dublin Core:

```
<book year="1994" xmlns:dcx="http://purl.org/dc/elements/1.1/">
    <dcx:title>TCP/IP Illustrated</dcx:title>
    <author><last>Stevens</last><first>W.</first></author>
</book>
```

In this data, `xmlns:dcx="http://purl.org/dc/elements/1.1/"` declares the prefix `"dcx"` as a synonym for the full namespace, and the element name `dcx:title` uses the prefix to indicate this is a `title` element as defined in the Dublin Core. The following query finds Dublin Core titles:

```
declare namespace dc="http://purl.org/dc/elements/1.1/"
doc("books.xml")//dc:title
```

The first line declares the namespace `dc` as a synonym for the Dublin Core namespace. Note that the prefix used in the document differs from the prefix used in the query. In XQuery, the name used for comparisons consists of the namespace URI and the "local part," which is `title` for this element.

Wildcards allow queries to select elements or attributes without specifying their entire names. For instance, a query might want to return all the elements of a given book, without specifying each possible element by name. In XQuery, this can be done with the following query:

```
doc("books.xml")//book[1]/*
```

The `*` wildcard matches any element, whether or not it is in a namespace. To match any attribute, use `@*`. To match any name in the namespace associated with the `dc` prefix, use `dc:*`. To match any `title` element, regardless of namespace, use `*:title`.

Creating Nodes: Element, Attribute, and Document Constructors

In the last section, we learned how to locate nodes in XML documents. Now we will learn how to create nodes. Elements, attributes, text nodes, processing instructions, and comments can all be created using the same syntax as XML. For instance, here is an element constructor that creates a book:

```
<book year="1977">
    <title>Harold and the Purple Crayon</title>
    <author><last>Johnson</last><first>Crockett</first></author>
    <publisher>HarperCollins Juvenile Books</publisher>
    <price>14.95</price>
</book>
```

As we have mentioned previously, the document node does not have explicit syntax in XML, but XQuery provides an explicit document node **constructor.** The query `document { }` creates an empty document node. Let's use a document node constructor together with other constructors to create an entire document, including the document node, a processing instruction for stylesheet linking, and an XML comment:

```
document {
  <?xml-stylesheet type="text/xsl"
                   href="c:\temp\double-slash.xslt"?>,
  <!—I love this book! —>,
  <book year="1977">
    <title>Harold and the Purple Crayon</title>
    <author><last>Johnson</last><first>Crockett</first></author>
    <publisher>HarperCollins Juvenile Books</publisher>
    <price>14.95</price>
  </book>
}
```

Constructors can be combined with other XQuery expressions to generate content dynamically. In an element constructor, curly braces, { }, delimit enclosed expressions, which are evaluated to create open content. Enclosed expressions may occur in the content of an element or the value of an attribute. For instance, the following query might be used in an interactive XQuery tutorial to teach how element constructors work:

```
<example>
   <p> Here is a query. </p>
   <eg> doc("books.xml")//book[1]/title </eg>
   <p> Here is the result of the above query.</p>
   <eg>{ doc("books.xml")//book[1]/title }</eg>
</example>
```

Here is the result of executing the above query for our sample data:

```
<example>
   <p> Here is a query. </p>
   <eg> doc("books.xml")//book[1]/title </eg>
   <p> Here is the result of the above query.</p>
   <eg><title>TCP/IP Illustrated</title></eg>
</example>
```

Enclosed expressions in element constructors permit new XML values to be created by restructuring existing XML values. Here is a query that creates a list of `book` titles from the bibliography:

```
<titles count="{ count(doc('books.xml')//title) }">
 {
  doc("books.xml")//title
 }
</titles>
```

The output of this query follows:

```
<titles count = "4">
   <title>TCP/IP Illustrated</title>
   <title>Advanced Programming in the Unix Environment</title>
   <title>Data on the Web</title>
   <title>The Economics of Technology and Content for
   Digital TV</title>
</titles>
```

Namespace declaration attributes in element constructors have the same meaning they have in XML. We previously showed the following Dublin Core example as XML text—but it is equally valid as an XQuery element constructor, and it treats the namespace declaration the same way:

```
<book year="1994" xmlns:dcx="http://purl.org/dc/elements/1.1/">
    <dcx:title>TCP/IP Illustrated</dcx:title>
    <author><last>Stevens</last><first>W.</first></author>
</book>
```

Computed element and attribute constructors are an alternative syntax that can be used as the XML-style constructors are, but they offer additional functionality that is discussed in this section. Here is a computed element constructor that creates an element named `title`, with the content `"Harold and the Purple Crayon"`. Inside the curly braces, constants are represented using XQuery's native syntax, in which strings are delimited by double or single quotes.

```
element title {
   "Harold and the Purple Crayon"
}
```

Here is a slightly more complex constructor that creates nested elements and attributes using the computed constructor syntax:

```
element book
{
    attribute year { 1977 },
    element author
    {
        element first { "Crockett" },
        element last { "Johnson" }
    },
        element publisher {"HarperCollins Juvenile Books"},
        element price { 14.95 }
}
```

The preceding example uses literals for the names of elements. In a computed element or attribute constructor, the name can also be an enclosed expression that must have the type *QName*, which represents an element or attribute name. For instance, suppose the user has written a function that takes two parameters, an element name in English and a language, and returns a *QName* that has been translated to the desired language. This function could be used in a computed element constructor as follows:

```
element { translate-element-name("publisher", "German") }
        { "HarperCollins Juvenile Books" }
```

The result of the above query is

```
<Verlag>HarperCollins Juvenile Books</Verlag>
```

In constructors, if sequences of **whitespace** characters occur in the boundaries between **tags** or enclosed expressions, with no intervening non-whitespace characters, then the whitespace is known as boundary whitespace. Implementations may discard boundary whitespace unless the query specifically declares that space must be preserved using the `xmlspace` declaration, a declaration that can occur in the **prolog.** The following query declares that all whitespace in element constructors must be preserved:

```
declare xmlspace = preserve

<author>
    <last>Stevens</last>
    <first>W.</first>
</author>
```

The output of the above query is

```
<author>
    <last>Stevens</last>
    <first>W.</first>
</author>
```

If the `xmlspace` declaration is absent, or is set to `strip`, then boundary whitespace is stripped:

```
<author><last>Stevens</last><first>W.</first></author>
```

Combining and Restructuring Nodes

Queries in XQuery often combine information from one or more sources and restructure it to create a new result. This section focuses on the expressions and functions most commonly used for combining and restructuring XML data.

FLWOR Expressions

FLWOR expressions, pronounced "flower expressions," are one of the most powerful and common expressions in XQuery. They are similar to the `SELECT-FROM-WHERE` statements in **SQL.** However, a FLWOR expression is not defined in terms of tables, rows, and columns; instead, a FLWOR expression binds variables to values in `for` and `let` clauses, and uses these variable bindings to create new results. A combination of variable bindings created by the `for` and `let` clauses of a FLWOR expression is called a tuple.

For instance, here is a simple FLWOR expression that returns the title and price of each book that was published in the year 2000:

```
for $b in doc("books.xml")//book
where $b/@year = "2000"
return $b/title
```

This query binds the variable `$b` to each book, one at a time, to create a series of tuples. Each tuple contains one variable binding in which `$b` is bound to a single book. The `where` clause tests each tuple to see if `$b/@year` is equal to "2000," and the `return` clause is evaluated for each tuple that satisfies the conditions expressed in the `where` clause. In our sample data, only *Data on the Web* was written in 2000, so the result of this query is

```
<title>Data on the Web</title>
```

The name FLWOR is an acronym, standing for the first letter of the clauses that may occur in a FLWOR expression:

- `for` clauses: associate one or more variables to expressions, creating a tuple stream in which each tuple binds a given variable to one of the items to which its associated expression evaluates
- `let` clauses: bind variables to the entire result of an expression, adding these bindings to the tuples generated by a `for` clause, or creating a single tuple to contain these bindings if there is no `for` clause
- `where` clauses: filter tuples, retaining only those tuples that satisfy a condition
- `order by` clauses: sort the tuples in a tuple stream
- `return` clauses: build the result of the FLWOR expression for a given tuple

The acronym FLWOR roughly follows the order in which the clauses occur. A FLWOR expression starts with one or more `for` or `let` clauses in any order, followed by an optional `where` clause, an optional `order by` clause, and a required `return` clause.

The `for` *and* `let` *Clauses*

Every clause in a FLWOR expression is defined in terms of tuples, and the `for` and `let` clauses create the tuples. Therefore, every FLWOR expression must have at least one `for` or `let` clause. It is extremely important to understand how tuples are generated in FLWOR expressions, so we will start with a series of artificial queries that show this in detail for various combinations of `for` clauses and `let` clauses.

We have already shown an example that binds one variable in a `for` clause. The following query creates an element named `tuple` in its `return` clause to show the tuples generated by such a query:

```
for $i in (1, 2, 3)
return
   <tuple><i>{ $i }</i></tuple>
```

In this example, we bind `$i` to the expression `(1, 2, 3)`, which constructs a sequence of integers. XQuery has a very general syntax, and `for` clauses or `let` clauses can be bound to any XQuery expression. Here is the result of the above query, showing how the variable `$i` is bound in each tuple:

```
<tuple><i>1</i></tuple>
<tuple><i>2</i></tuple>
<tuple><i>3</i></tuple>
```

Note that the order of the items bound in the tuple is the same as the order of the items in the original expression `(1, 2, 3)`. A `for` clause preserves order when it creates tuples.

A `let` clause binds a variable to the entire result of an expression. If there are no `for` clauses in the FLWOR expression, then a single tuple is created, containing the variable bindings from the `let` clauses. The following query is like the previous query, but it uses a `let` clause rather than a `for`:

```
let $i := (1, 2, 3)
return
   <tuple><i>{ $i }</i></tuple>
```

The result of this query contains only one tuple, in which the variable `$i` is bound to the entire sequence of integers:

```
<tuple><i>1 2 3</i></tuple>
```

If a `let` clause is used in a FLWOR expression that has one or more `for` clauses, the variable bindings of `let` clauses are added to the tuples generated by the `for` clauses. This is demonstrated by the following query:

```
for $i in (1, 2, 3)
let $j := (1, 2, 3)
return
   <tuple><i>{ $i }</i><j>{ $j }</j></tuple>
```

If a `let` clause is used in a FLWOR expression that has one or more `for` clauses, the variable bindings from `let` clauses are added to the tuples generated by the `for` clauses:

```
  <tuple><i>1</i><j>1 2 3</j></tuple>
  <tuple><i>2</i><j>1 2 3</j></tuple>
  <tuple><i>3</i><j>1 2 3</j></tuple>
```

Here is a query that combines `for` and `let` clauses in the same way as the previous query:

```
for $b in doc("books.xml")//book
let $c := $b/author
return <book>{ $b/title, <count>{ count($c) }</count>}</book>
```

This query lists the title of each book together with the number of authors. Listing 1.3 shows the result when we apply it to our bibliography data.

Listing 1.3 Query Results

```
<book>
  <title>TCP/IP Illustrated</title>
  <count>1</count>
</book>
<book>
  <title>Advanced Programming in the UNIX Environment</title>
  <count>1</count>
</book>
<book>
  <title>Data on the Web</title>
  <count>3</count>
</book>
<book>
  <title>The Economics of Technology and Content for
  Digital TV</title>
  <count>0</count>
</book>
```

If more than one variable is bound in the `for` clauses of a FLWOR expression, then the tuples contain all possible combinations of the items to which these variables are bound. For instance, the following query shows all combinations that include 1, 2, or 3 combined with 4, 5, or 6:

```
for $i in (1, 2, 3),
    $j in (4, 5, 6)
return
   <tuple><i>{ $i }</i><j>{ $j }</j></tuple>
```

Here is the result of the above query:

```
<tuple><i>1</i><j>4</j></tuple>
<tuple><i>1</i><j>5</j></tuple>
<tuple><i>1</i><j>6</j></tuple>
<tuple><i>2</i><j>4</j></tuple>
<tuple><i>2</i><j>5</j></tuple>
<tuple><i>2</i><j>6</j></tuple>
<tuple><i>3</i><j>4</j></tuple>
<tuple><i>3</i><j>5</j></tuple>
<tuple><i>3</i><j>6</j></tuple>
```

A combination of all possible combinations of sets of values is called a Cartesian cross-product. The tuples preserve the order of the original sequences, in the order in which they are bound. In the previous example, note that the tuples reflect the values of each `$i` in the original order; for a given value of `$i`, the values of `$j` occur in the original order. In mathematical terms, the tuples generated in a FLWOR expression are drawn from the ordered Cartesian cross-product of the items to which the `for` variables are bound.

The ability to create tuples that reflect combinations becomes particularly interesting when combined with `where` clauses to perform **joins.** The following sections illustrate this in depth. But first we must introduce the `where` and `return` clauses.

The `where` *Clause*

A `where` clause eliminates tuples that do not satisfy a particular condition. A `return` clause is only evaluated for tuples that survive the `where` clause. The following query returns only books whose prices are less than $50.00:

```
for $b in doc("books.xml")//book
where $b/price < 50.00
return $b/title
```

Here is the result of this query:

```
  <title>Data on the Web</title>
```

A `where` clause can contain any expression that evaluates to a Boolean value. In SQL, a `WHERE` clause can only test single values, but there is no such restriction on `where` clauses in XQuery. The following query returns the title of books that have more than two authors:

```
for $b in doc("books.xml")//book
let $c := $b//author
where count($c) > 2
return $b/title
```

Here is the result of the above query:

```
<title>Data on the Web</title>
```

The `order by` *Clause*

The `order by` clause sorts the tuples before the `return` clause is evaluated in order to change the order of results. For instance, the following query lists the titles of books in alphabetical order:

```
for $t in doc("books.xml")//title
order by $t
return $t
```

The `for` clause generates a sequence of tuples, with one `title` node in each tuple. The `order by` clause sorts these tuples according to the value of the `title` elements in the tuples, and the `return` clause returns the `title` elements in the same order as the sorted tuples. The result of this query is

```
<title>Advanced Programming in the Unix Environment</title>
<title>Data on the Web</title>
<title>TCP/IP Illustrated</title>
<title>The Economics of Technology and Content for Digital TV</title>
```

The `order by` clause allows one or more orderspecs, each of which specifies one expression used to sort the tuples. An **orderspec** may also specify whether to sort in ascending or descending order, how expressions that evaluate to empty sequences should be sorted, a specific collation to be used, and whether stable sorting should be used (stable sorting preserves the relative order of two items if their values are equal). Here is a query that returns authors, sorting in reverse order by the last name, then the first name:

```
for $a in doc("books.xml")//author
order by $a/last descending, $a/first descending
return $a
```

The result of this query is shown in Listing 1.4.

Listing 1.4 Results of Query for Authors Sorted by Last Name

```
<author>
    <last>Suciu</last>
    <first>Dan</first>
</author>
<author>
    <last>Stevens</last>
    <first>W.</first>
</author>
<author>
    <last>Stevens</last>
    <first>W.</first>
</author>
<author>
    <last>Buneman</last>
    <first>Peter</first>
</author>
<author>
    <last>Abiteboul</last>
    <first>Serge</first>
</author>
```

The `order by` clause may specify conditions based on data that is not used in the `return` clause, so there is no need for an expression to return data in order to use it to sort. Here is an example that returns the titles of books, sorted by the name of the first author:

```
let $b := doc("books.xml")//book
for $t in distinct-values($b/title)
let $a1 := $b[title=$t]/author[1]
order by $a1/last, $a1/first
return $b/title
```

The result of this query is

```
<title>The Economics of Technology and Content for Digital TV</title>
<title>Data on the Web</title>
<title>Advanced Programming in the UNIX Environment</title>
<title>TCP/IP Illustrated</title>
```

The first book in this list has editors, but no authors. For this book, `$a1/last` and `$a1/first` will both return empty sequences. Some XQuery implementations always sort empty sequences as the greatest possible value; others always sort empty sequences as the least possible value. The XML Query Working Group decided to allow vendors to choose which of these orders to implement because many XQuery implementations present views of relational data, and relational databases differ in their sorting of **nulls**. To guarantee that an XQuery uses the same sort order across implementations, specify "empty greatest" or "empty least" in an orderspec if its expression can evaluate to an empty sequence.

Two books in our data are written by the same author, and we may want to ensure that the original order of these two books is maintained. We can do this by specifying a stable sort, which maintains the relative order of two items if the comparison expressions consider them equal. The following query specifies a stable sort, and requires empty sequences to be sorted as least:

```
let $b := doc("books.xml")//book
for $t in distinct-values($b/title)
let $a1 := $b[title=$t]/author[1]
stable order by $a1/last empty least, $a1/first empty least
return $b/title
```

This query returns the same result as the previous one, but is guaranteed to do so across all implementations.

Collations may also be specified in an `order by` clause. The following query sorts titles using a U.S. English collation:

```
for $t in doc("books.xml")//title
order by $t collation "http://www.example.com/collations/eng-us"
return $t
```

Most queries use the same collation for all comparisons, and it is generally too tedious to specify a collation for every orderspec. XQuery allows a default collation to be specified in the prolog. The default collation is used when the orderspec does not specify a collation. Here is a query that sets http://www.example.com/collations/eng-us as the default collation; it returns the same results as the previous query:

```
default collation = "http://www.example.com/collations/eng-us"

for $t in doc("books.xml")//title
order by $t
return $t
```

When sorting expressions in queries, it is important to remember that the `/` and `//` operators sort in document order. That means that an order established with an `order by` clause can be changed by expressions that use these operators. For instance, consider the following query:

```
let $authors := for $a in doc("books.xml")//author
                order by $a/last, $a/first
                return $a
return $authors/last
```

This query does not return the author's last names in alphabetical order, because the `/` in `$authors/last` sorts the last elements in document order. This kind of error generally occurs with `let` bindings, not with `for` bindings, because a `for` clause binds each variable to a single value in a given tuple, and returning children or descendents of a single node does not lead to surprises. The following query returns author's last names in alphabetical order:

```
for $a in doc("books.xml")//author
order by $a/last, $a/first
return $a/last
```

The `return` *Clause*

We have already seen that a `for` clause or a `let` clause may be bound to any expression, and a `where` clause may contain any Boolean expression. Similary, any XQuery expression may occur in a `return` clause. Element constructors are an extremely common expression in `return` clauses; for instance, the following query uses an element constructor to create price quotes:

```
for $b in doc("books.xml")//book
return
  <quote>{ $b/title, $b/price }</quote>
```

Listing 1.5 shows the result of the above query.

Listing 1.5 Results of Query for Price Quotes

```
<quote>
    <title>TCP/IP Illustrated</title>
    <price>65.95</price>
</quote>
<quote>
    <title>Advanced Programming in the UNIX Environment</title>
    <price>65.95</price>
</quote>
```

Listing 1.5 Results of Query for Price Quotes *(continued)*

```
<quote>
    <title>Data on the Web</title>
    <price>39.95</price>
</quote>
<quote>
    <title>The Economics of Technology and Content for Digital
TV</title>
    <price>129.95</price>
</quote>
```

Element constructors can be used in a `return` clause to change the hierarchy of data. For instance, we might want to represent an author's name as a string in a single element, which we can do with the following query:

```
for $a in doc("books.xml")//author
return
  <author>{ string($a/first), " ", string($a/last) }</author>
```

Here is the result of the above query:

```
<author>W. Stevens</author>
<author>W. Stevens</author>
<author>Serge Abiteboul</author>
<author>Peter Buneman</author>
<author>Dan Suciu</author>
```

Another application might want to insert a name element to hold the first and last name of the author—after all, an author does not consist of a first and a last! Here is a query that adds a level to the hierarchy for names:

```
for $a in doc("books.xml")//author
return
<author>
   <name>{ $a/first, $a/last }</name>
  </author>
```

Here is one author's name taken from the output of the above query:

```
<author>
    <name>
    <first>Serge</first>
    <last>Abiteboul</last>
    </name>
</author>
```

This section has discussed the most straightforward use of `for` and `return` clauses, and it has shown how to combine FLWOR expressions with other expressions to perform common tasks. More complex uses of `for` clauses are explored later in separate sections on joins and positional variables.

The Positional Variable `at`

The `for` clause supports positional variables, which identify the position of a given item in the expression that generated it. For instance, the following query returns the titles of books, with an attribute that numbers the books:

```
for $t at $i in doc("books.xml")//title
return <title pos="{$i}">{string($t)}</title>
```

Here is the result of this query:

```
<title pos="1">TCP/IP Illustrated</title>
<title pos="2">Advanced Programming in the Unix Environment</title>
<title pos="3">Data on the Web</title>
<title pos="4">The Economics of Technology and Content for Digital
TV</title>
```

In some data, position conveys meaning. In tables, for instance, the row and column in which an item is found often determine its meaning. For instance, suppose we wanted to create data from an XHTML web page that contains the table shown in Table 1.2.

TABLE 1.2 Table from an XHTML Web Page

Title	**Publisher**	**Price**	**Year**
TCP/IP Illustrated	Addison-Wesley	65.95	1994
Advanced Programming in the UNIX Environment	Addison-Wesley	65.95	1992
Data on the Web	Morgan Kaufmann Publishers	39.95	2000
The Economics of Technology and Content for Digital TV	Kluwer Academic Publishers	129.95	1999

The XHTML source for this table is shown in Listing 1.2.

Listing 1.6 XHTML Source for Table 1.2

```
<table border="1">
   <thead>
     <tr>
           <td>Title</td>
           <td>Publisher</td>
           <td>Price</td>
           <td>Year</td>
     </tr>
   </thead>
   <tbody>
     <tr>
           <td>TCP/IP Illustrated</td>
           <td>Addison-Wesley</td>
           <td>65.95</td>
           <td>1994</td>
     </tr>
     <tr>
           <td>Advanced Programming in the UNIX
           Environment</td>
           <td>Addison-Wesley</td>
           <td>65.95</td>
           <td>1992</td>
     </tr>
  <!— Additional rows omitted to save space —>
   </tbody>
</table>
```

In this table, every entry in the same column as the `Title` header is a title, every entry in the same column as the `Publisher` header is a publisher, and so forth. In other words, we can determine the purpose of an entry if we can determine its position as a column of the table, and relate it to the position of a column header. Positional variables make this possible. Since XHTML is XML, it can be queried using XQuery. Listing 1.7 shows a query that produces meaningful XML from the above data, generating the names of elements from the column headers.

Listing 1.7 Query to Generate Names of Elements from Column Headers

```
let $t := doc("bib.xhtml")//table[1]
for $r in $t/tbody/tr
return
  <book>
```

Listing 1.7 Query to Generate Names of Elements from Column Headers *(continued)*

```
   {
     for $c at $i in $r/td
     return element{ lower-case(data($t/thead/tr/td[$i])) }
                   { string( $c) }
   }
  </book>
```

Note the use of a computed element constructor that uses the column header to determine the name of the element. Listing 1.8 shows the portion of the output this query generates for the partial data shown in Table 1.2.

Listing 1.8 Output Generated by the Query of Listing 1.7

```
<book>
   <title>TCP/IP Illustrated</title>
   <publisher>Addison-Wesley</publisher>
   <price>65.95</price>
   <year>1994</year>
</book>
<book>
   <title>Advanced Programming in the Unix Environment</title>
   <publisher>Addison-Wesley</publisher>
   <price>65.95</price>
   <year>1992</year>
</book>
```

Eliminating Duplicate Subtrees with `distinct-values()` *and FLWOR Expressions*

Data often contains duplicate values, and FLWOR expressions are often combined with the `distinct-values()` function to remove duplicates from subtrees. Let's start with the following query, which returns the last name of each author:

```
doc("books.xml")//author/last
```

Since one of our authors wrote two of the books in the bibliography, the result of this query contains a duplicate:

```
<last>Stevens</last>
<last>Stevens</last>
<last>Abiteboul</last>
<last>Buneman</last>
<last>Suciu</last>
```

The `distinct-values()` function extracts the values of a sequence of nodes and creates a sequence of unique values, eliminating duplicates. Here is a query that uses `distinct-values()` to eliminate duplicate last names:

```
distinct-values(doc("books.xml")//author/last)
```

Here is the output of the above query:

```
Stevens Abiteboul Buneman Suciu
```

The `distinct-values()` function eliminates duplicates, but in order to do so, it extracts values from nodes. FLWOR expressions are often used together with `distinct-values()` to create subtrees that correspond to sets of one or more unique values. For the preceding query, we can use an element constructor to create a last element containing each value:

```
for $l in distinct-values(doc("books.xml")//author/last)
return <last>{ $l }</last>
```

Here is the output of the above query:

```
<last>Stevens</last>
<last>Abiteboul</last>
<last>Buneman</last>
<last>Suciu</last>
```

The same problem arises for complex subtrees. For instance, the following query returns authors, and one of the authors is a duplicate by both first and last name:

```
doc("books.xml")//author
```

The output of the above query appears in Listing 1.9.

Listing 1.9 Output of the Query for Authors

```
<authors>
    <author>
      <last>Stevens</last>
      <first>W.</first>
    </author>
    <author>
      <last>Stevens</last>
      <first>W.</first>
    </author>
```

Katz_C01.qxd 7/23/03 2:45 PM Page 32

Listing 1.9 Output of the Query for Authors *(continued)*

```
    <author>
      <last>Abiteboul</last>
      <first>Serge</first>
    </author>
    <author>
      <last>Buneman</last>
      <first>Peter</first>
    </author>
    <author>
      <last>Suciu</last>
      <first>Dan</first>
    </author>
  </authors>
```

To eliminate duplicates from complex subtrees, we have to decide what criterion to use for detecting a duplicate. In this case, let's say that an author is a duplicate if there is another author who has the same first and last names. Now let's write a query that returns one author for each first and last name that occur together within an author element in our dataset:

```
let $a := doc("books.xml")//author
for $l in distinct-values($a/last),
    $f in distinct-values($a[last=$l]/first)
return
    <author>
      <last>{ $l }</last>
      <first>{ $f }</first>
    </author>
```

In the output of the above query (Listing 1.10), each author's name appears only once.

Listing 1.10 Output of Query to Avoid Duplicate Author Names

```
<authors>
    <author>
      <last>Stevens</last>
      <first>W.</first>
    </author>
    <author>
      <last>Abiteboul</last>
      <first>Serge</first>
    </author>
    <author>
      <last>Buneman</last>
      <first>Peter</first>
    </author>
```

Listing 1.10 Output of Query to Avoid Duplicate Author Names *(continued)*

```
    <author>
      <last>Suciu</last>
      <first>Dan</first>
    </author>
  </authors>
```

Joins: Combining Data Sources with `for` *and* `where` *Clauses*

A query may bind multiple variables in a `for` clause in order to combine information from different expressions. This is often done to bring together information from different data sources. For instance, suppose we have a file named `reviews.xml` that contains book reviews:

```
<reviews>
  <entry>
   <title>TCP/IP Illustrated</title>
   <rating>5</rating>
   <remarks>Excellent technical content. Not much plot.</remarks>
  </entry>
</reviews>
```

A FLWOR expression can bind one variable to our bibliography data and another to the reviews, making it possible to compare data from both files and to create results that combine their information. For instance, a query could return the title of a book and any remarks found in a review.

As we have discussed earlier, the Cartesian cross-product of two sequences contains all possible combinations of the items in those sequences. When a `where` clause is used to select interesting combinations from the Cartesian cross-product, this is known as a join. The following query performs a join to combine data from a bibliography with data from a set of reviews:

```
for $t in doc("books.xml")//title,
    $e in doc("reviews.xml")//entry
where $t = $e/title
return <review>{ $t, $e/remarks }</review>
```

The result of this query is as follows:

```
<review>
    <title>TCP/IP Illustrated</title>
    <remarks>Excellent technical content. Not much plot.</remarks>
</review>
```

In this query, the `for` clauses create tuples from the Cartesian cross-product of titles and entries, the `where` clause filters out tuples where the title of the review does not match the title of the book, and the `return` clause constructs the result from the remaining tuples. Note that only books with reviews are shown. SQL programmers will recognize the preceding query as an inner join, returning combinations of data that satisfy a condition.

The tuples generated for a FLWOR expression include all expressions bound in variable bindings in `for` clauses. A FLWOR expression with multiple `for` clauses has the same meaning as a FLWOR expression that binds multiple variables in a single `for` clause. The following query is precisely equivalent to the previous one:

```
for $t in doc("books.xml")//title
for $e in doc("reviews.xml")//entry
where $t = $e/title
return <review>{ $t, $e/remarks }</review>
```

The query shown in Listing 1.11 returns the title of each book regardless of whether it has a review; if a book does have a review, the remarks found in the review are also included. SQL programmers will recognize this as a **left outer join**.

Listing 1.11 Query to Return Titles with or without Reviews

```
for $t in doc("books.xml")//title
return
  <review>
   { $t }
   {
     for $e in doc("reviews.xml")//entry
     where $e/title = $t
     return $e/remarks
   }
  </review>
```

Inverting Hierarchies

XQuery can be used to do quite general transformations. One transformation that is used in many applications is colloquially referred to as “inverting a hierarchy”—creating a new document in which the top nodes represent information which was found in the lower nodes of the original document. For instance, in our sample data, publishers are found at the bottom of the hierarchy, and books are found near the top. Listing 1.12 shows a query that creates a list of titles published by each publisher, placing the publisher at the top of the hierarchy and listing the titles of books at the bottom.

Listing 1.12 Query to List Titles by Publisher

```
<listings>
  {
    for $p in distinct-values(doc("books.xml")//publisher)
    order by $p
    return
        <result>
            { $p }
            {
                for $b in doc("books.xml")/bib/book
                where $b/publisher = $p
                order by $b/title
                return $b/title
            }
        </result>
  }
</listings>
```

```
The results of this query are as follows:
```

```
<listings>
 <result>
     <publisher>Addison-Wesley</publisher>
   <title>Advanced Programming in the Unix Environment</title>
   <title>TCP/IP Illustrated</title>
    </result>
    <result>
   <publisher>Kluwer Academic Publishers</publisher>
   <title>The Economics of Technology and Content for
   Digital TV</title>
    </result>
    <result>
   <publisher>Morgan Kaufmann Publishers</publisher>
   <title>Data on the Web</title>
    </result>
</listings>
```

A more complex example of inverting a hierarchy is discussed in the following section on quantifiers.

Quantifiers

Some queries need to determine whether at least one item in a sequence satisfies a condition, or whether every item in a sequence satisfies a condition. This is done using quantifiers. An existential quantifier tests whether at least one item satisfies a condition. The following query shows an existential quantifier in XQuery:

```
for $b in doc("books.xml")//book
where some $a in $b/author
      satisfies ($a/last="Stevens" and $a/first="W.")
return $b/title
```

The `some` quantifier in the `where` clause tests to see if there is at least one author that satisfies the conditions given inside the parentheses. Here is the result of the above query:

```
<title>TCP/IP Illustrated</title>
<title>Advanced Programming in the Unix Environment</title>
```

A universal quantifier tests whether every node in a sequence satisfies a condition. The following query tests to see if every author of a book is named W. Stevens:

```
for $b in doc("books.xml")//book
where every $a in $b/author
      satisfies ($a/last="Stevens" and $a/first="W.")
return $b/title
```

Here is the result of the above query:

```
<title>TCP/IP Illustrated</title>
<title>Advanced Programming in the Unix Environment</title>
<title>The Economics of Technology and Content for Digital TV</title>
```

The last title returned, *The Economics of Technology and Content for Digital TV*, is the title of a book that has editors but no authors. For this book, the expression `$b/author` evaluates to an empty sequence. If a universal quantifier is applied to an empty sequence, it always returns true, because every item in that (empty) sequence satisfies the condition—even though there are no items.

Quantifiers sometimes make complex queries much easier to write and understand. For instance, they are often useful in queries that invert hierarchies. Listing 1.13 shows a query that creates a list of books written by each author in our bibliography.

Listing 1.13 Query to List Books by Author

```
<author-list>
  {
    let $a := doc("books.xml")//author
    for $l in distinct-values($a/last),
        $f in distinct-values($a[last=$l]/first)
    order by $l, $f
    return
        <author>
          <name>{ $l, ", ", $f }</name>
          {
             for $b in doc("books.xml")/bib/book
             where some $ba in $b/author satisfies
                   ($ba/last=$l and $ba/first=$f)
             order by $b/title
             return $b/title
          }
        </author>
  }
</author-list>
```

The result of the above query is shown in Listing 1.14.

Listing 1.14 Results of Query to List Books by Author

```
<author-list>
    <author>
        <name>Stevens, W.</name>
        <title>Advanced Programming in the Unix Environment</title>
        <title>TCP/IP Illustrated</title>
    </author>
    <author>
        <name>Abiteboul, Serge</name>
        <title>Data on the Web</title>
    </author>
    <author>
        <name>Buneman, Peter</name>
        <title>Data on the Web</title>
    </author>
    <author>
        <name>Suciu, Dan</name>
        <title>Data on the Web</title>
    </author>
</author-list>
```

Conditional Expressions

XQuery's conditional expressions are used in the same way as conditional expressions in other languages. Listing 1.15 shows a query that uses a conditional expression to list the first two authors' names for each book and a dummy name containing "et al." to represent any remaining authors.

Listing 1.15 Query to List Author's Names with "et al."

```
for $b in doc("books.xml")//book
return
  <book>
    { $b/title }
    {
      for $a at $i in $b/author
      where $i <= 2
      return <author>{string($a/last), ", ",
                     string($a/first)}</author>
    }
    {
      if (count($b/author) > 2)
      then <author>et al.</author>
      else ()
    }
   </book>
```

In XQuery, both the `then` clause and the `if` clause are required. Note that the empty sequence `()` can be used to specify that a clause should return nothing. The output of this query is shown in Listing 1.16.

Listing 1.16 Result of Query from Listing 1.15

```
<book>
    <title>TCP/IP Illustrated</title>
    <author>Stevens, W.</author>
</book>
<book>
    <title>Advanced Programming in the Unix Environment</title>
    <author>Stevens, W.</author>
</book>
<book>
    <title>Data on the Web</title>
    <author>Abiteboul, Serge</author>
    <author>Buneman, Peter</author>
    <author>et al.</author>
</book>
```

Listing 1.16 Result of Query from Listing 1.15 *(continued)*

```
<book>
    <title>The Economics of Technology and Content for
    Digital TV</title>
</book>
```

Operators

The queries we have shown up to now all contain operators, which we have not yet covered. Like most languages, XQuery has arithmetic operators and comparison operators, and because sequences of nodes are a fundamental datatype in XQuery, it is not surprising that XQuery also has node sequence operators. This section describes these operators in some detail. In particular, it describes how XQuery treats some of the cases that arise quite easily when processing XML; for instance, consider the following expression: `1 * $b`. How is this interpreted if `$b` is an empty sequence, untyped character data, an element, or a sequence of five nodes? Given the flexible structure of XML, it is imperative that cases like this be well defined in the language. (Chapter 2, "Influences on the Design of XQuery," provides additional background on the technical complexities that the working group had to deal with to resolve these and similar issues.)

Two basic operations are central to the use of operators and functions in XQuery. The first is called **typed value** extraction. We have already used typed value extraction in many of our queries, without commenting on it. For instance, we have seen this query:

```
doc("books.xml")/bib/book/author[last='Stevens']
```

Consider the expression `last='Stevens'`. If `last` is an element, and `'Stevens'` is a string, how can an element and a string be equal? The answer is that the `=` operator extracts the typed value of the element, resulting in a **string value** that is then compared to the string `Stevens`. If the document is governed by a W3C XML Schema, then it may be associated with a simple type, such as `xs:integer`. If so, the typed value will have whatever type has been assigned to the node by the schema. XQuery has a function called `data()` that extracts the typed value of a function. Assuming the following element has been validated by a schema processor, the result of this query is the integer 4:

```
data( <e xsi:type="xs:integer">4</e> )
```

A query may import a schema. We will discuss **schema imports** later, but schema imports have one effect that should be understood now. If typed value extraction is applied to an element, and the query has imported a schema definition for that element specifying that the element may have other elements as children, then typed value extraction raises an error.

Typed value extraction is defined for a single item. The more general form of typed value extraction is called **atomization,** which defines how typed value extraction is done for any sequence of items. For instance, atomization would be performed for the following query:

```
avg( 1, <e>2</e>, <e xsi:type="xs:integer">3</e> )
```

Atomization simply returns the typed value of every item in the sequence. The preceding query returns 2, which is the average of 1, 2, and 3. In XQuery, atomization is used for the operands of arithmetic expressions and comparison expressions. It is also used for the parameters and return values of functions and for cast expressions, which are discussed in other sections.

Arithmetic Operators

XQuery supports the arithmetic operators `+`, `-`, `*`, `div`, `idiv`, and `mod`. The `div` operator performs division on any numeric type. The `idiv` operator requires integer arguments, and returns an integer as a result, rounding toward 0. All other arithmetic operators have their conventional meanings. If an operand of an arithmetic operator is a node, atomization is applied. For instance, the following query returns the integer 4:

```
2 + <int>{ 2 }</int>
```

If an operand is an empty sequence, the result of an arithmetic operator is an empty sequence. Empty sequences in XQuery frequently operate like nulls in SQL. The result of the following query is an empty sequence:

```
2 + ()
```

If an operand is **untyped data,** it is cast to a double, raising an error if the cast fails. This implicit cast is important, because a great deal of XML data is found in documents that do not use W3C XML Schema, and therefore do not have simple or complex types. Many of these documents however contain data that is to be interpreted as numeric. The prices in our sample document are one example of this. The following query adds the first and second prices, returning the result as a double:

```
let $p := doc("books.xml")//price
return $p[1] + $p[2]
```

Comparison Operators

XQuery has several sets of comparison operators, including value comparisons, general comparisons, node comparisons, and order comparisons. Value comparisons and general comparisons are closely related; in fact, each **general comparison operator** combines an existential quantifier with a corresponding a **value comparison operator.** Table 1.3 shows the value comparison operator to which each general comparison operator corresponds.

The value comparisons compare two atomic values. If either operand is a node, atomization is used to convert it to an atomic value. For the comparison, if either operand is untyped, it is treated as a string. Here is a query that uses the `eq` operator:

```
for $b in doc("books.xml")//book
where $b/title eq "Data on the Web"
return $b/price
```

TABLE 1.3 Value Comparison Operators vs. General Comparison Operators

Value Comparison Operator	General Comparison Operator
`eq`	`=`
`ne`	`!=`
`lt`	`<`
`le`	`<=`
`gt`	`>`
`ge`	`>=`

Using value comparisons, strings can only be compared to other strings, which means that value comparisons are fairly strict about typing. If our data is governed by a DTD, then it does not use the W3C XML Schema simple types, so the price is untyped. Therefore, a cast is needed to cast `price` to a `decimal` in the following query:

```
for $b in doc("books.xml")//book
where xs:decimal($b/price) gt 100.00
return $b/title
```

If the data were governed by a W3C XML Schema that declared `price` to be a `decimal`, this cast would not have been necessary. In general, if the data you are querying is meant to be interpreted as typed data, but there are no types in the XML, value comparisons force your query to cast when doing comparisons—general comparisons are more loosely typed and do not require such casts. This problem does not arise if the data is meant to be interpreted as string data, or if it contains the appropriate types.

Like arithmetic operators, value comparisons treat empty sequences much like SQL nulls. If either operand is an empty sequence, a value comparison evaluates to the empty sequence. If an operand contains more than one item, then a value comparison raises an error. Here is an example of a query that raises an error:

```
for $b in doc("books.xml")//book
where $b/author/last eq "Stevens"
return $b/title
```

The reason for the error is that many books have multiple authors, so the expression `$b/author/last` returns multiple nodes. The following query uses `=`, the general comparison that corresponds to `eq`, to return books for which *any* author's last name is equal to Stevens:

```
for $b in doc("books.xml")//book
where $b/author/last = "Stevens"
return $b/title
```

There are two significant differences between value comparisons and general comparisons. The first is illustrated in the previous query. Like value comparisons, general comparisons apply atomization to both operands, but instead of requiring each operand to be a single atomic value, the result of this atomization may be a sequence of atomic values. The general comparison returns `true` if *any* value on the left matches *any* value on the right, using the appropriate comparison.

The second difference involves the treatment of untyped data—general comparisons try to cast to an appropriate "required type" to make the comparison work. This is illustrated by the following query:

```
for $b in doc("books.xml")//book
where $b/price = 100.00
return $b/title
```

In this query, 100.00 is a decimal, and the `=` operator casts the price to decimal as well. When a general comparison tests a pair of atomic values and one of these values is untyped, it examines the other atomic value to determine the required type to which it casts the untyped operand:

- If the other atomic value has a numeric type, the required type is `xs:double`.
- If the other atomic value is also untyped, the required type is `xs:string`.
- Otherwise, the required type is the **dynamic type** of the other atomic value. If the cast to the required type fails, a dynamic error is raised.

These conversion rules mean that comparisons done with general comparisons rarely need to cast when working with data that does not contain W3C XML Schema simple types. On the other hand, when working with strongly typed data, value comparisons offer greater type safety.

You should be careful when using the `=` operator when an operand has more than one step, because it can lead to confusing results. Consider the following query:

```
for $b in doc("books.xml")//book
where $b/author/first = "Serge"
  and $b/author/last = "Suciu"
return $b
```

The result of this query may be somewhat surprising, as Listing 1.17 shows.

Listing 1.17 Surprising 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>
```

Since this book does have an author whose first name is "Serge" and an author whose last name is "Suciu," the result of the query is correct, but it is surprising. The following query expresses what the author of the previous query probably intended:

```
for $b in doc("books.xml")//book,
    $a in $b/author
where $a/first="Serge"
  and $a/last="Suciu"
return $b
```

Comparisons using the = operator are not transitive. Consider the following query:

```
let $a := ( <first>Jonathan</first>, <last>Robie</last> ),
       $b := ( <first>Jonathan</first>, <last>Marsh</last> ),
       $c := ( <first>Rodney</first>, <last>Marsh</last> )
return
<out>
  <equals>{ $a = $b }</equals>
  <equals>{ $b = $c }</equals>
  <equals>{ $a = $c }</equals>
</out>
```

Remember that = returns `true` if there is a value on the left that matches a value on the right. The output of this query is as follows:

```
<out>
  <equals>True</equals>
  <equals>True</equals>
  <equals>False</equals>
</out>
```

Node comparisons determine whether two expressions evaluate to the same node. There are two node comparisons in XQuery, `is` and `is not`. The following query tests whether the most expensive book is also the book with the greatest number of authors and editors:

```
let $b1 := for $b in doc("books.xml")//book
           order by count($b/author) + count($b/editor)
           return $b
let $b2 := for $b in doc("books.xml")//book
           order by $b/price
           return $b
return $b1[last()] is $b2[last()]
```

This query also illustrates the `last()` function, which determines whether a node is the last node in the sequence; in other words, `$b1[last()]` returns the last node in `$b1`.

XQuery provides two operators that can be used to determine whether one node comes before or after another node in document order. These operators are generally most useful for data in which the order of elements is meaningful, as it is in many documents or tables. The operator `$a << $b` returns true if `$a` precedes `$b` in document order; `$a >> $b` returns true if `$a` follows `$b` in document order. For instance, the following query returns books where Abiteboul is an author, but is not listed as the first author:

```
for $b in doc("books.xml")//book
let $a := ($b/author)[1],
    $sa := ($b/author)[last="Abiteboul"]
where $a << $sa
return $b
```

In our sample data, there are no such books.

Sequence Operators

XQuery provides the `union`, `intersect`, and `except` operators for combining sequences of nodes. Each of these operators combines two sequences, returning a result sequence in document order. As we have discussed earlier, a sequence of nodes that is in document order, never contains the same node twice. If an operand contains an item that is not a node, an error is raised.

The `union` operator takes two node sequences and returns a sequence with all nodes found in the two input sequences. This operator has two lexical forms: `|` and `union`. Here is a query that uses the `|` operator to return a sorted list of last names for all authors or editors:

```
let $l := distinct-values(doc("books.xml")//(author | editor)/last)
order by $l
return <last>{ $l }</last>
```

Here is the result of the above query:

```
<last>Abiteboul</last>
<last>Buneman</last>
<last>Gerbarg</last>
<last>Stevens</last>
<last>Suciu</last>
```

The fact that the `union` operator always returns nodes in document order is sometimes quite useful. For instance, the following query sorts books based on the name of the first author or editor listed for the book:

```
for $b in doc("books.xml")//book
let $a1 := ($b/author union $b/editor)[1]
order by $a1/last, $a1/first
return $b
```

The `intersect` operator takes two node sequences as operands and returns a sequence containing all the nodes that occur in both operands. The `except` operator takes two node sequences as operands and returns a sequence containing all the nodes that occur in the first operand but not in the second operand. For instance, the following query returns a book with all of its children except for the price:

```
for $b in doc("books.xml")//book
where $b/title = "TCP/IP Illustrated"
return
   <book>
    { $b/@* }
    { $b/* except $b/price }
   </book>
```

The result of this query contains all attributes of the original book and all elements—in document order—except for the `price` element, which is omitted:

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

Built-in Functions

XQuery has a set of built-in functions and operators, including many that are familiar from other languages, and some that are used for customized XML processing. The complete list of built-in functions is found in [XQ-FO]. This section focuses on the most commonly used functions, some of which must be understood to follow what is said in the rest of the chapter.

SQL programmers will be familiar with the `min()`, `max()`, `count()`, `sum()`, and `avg()` functions. The following query returns the titles of books that are more expensive than the average book:

```
let $b := doc("books.xml")//book
let $avg := average( $b//price )
return $b[price > $avg]
```

For our sample data, Listing 1.18 shows the result of this query.

Listing 1.18 Result of Query for Books More Expensive Than Average

```
<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>
```

Note that price is the name of an element, but `max()` is defined for atomic values, not for elements. In XQuery, if the type of a function argument is an **atomic type,** then the following conversion rules are applied. If the argument is a node, its typed value is extracted, resulting in a sequence of values. If any value in the argument sequence is untyped, XQuery attempts to convert it to the required type and raises an error if it fails. A value is accepted if it has the expected type.

Other familiar functions in XQuery include numeric functions like `round()`, `floor()`, and `ceiling()`; string functions like `concat()`, `string-length()`, `starts-with()`, `ends-with()`, `substring()`, `upper-case()`, `lower-case()`; and casts for the various simple types. These are all covered in [XQ-FO], which defines the standard function library for XQuery; they need no further coverage here since they are straightforward.

XQuery also has a number of functions that are not found in most other languages. We have already covered `distinct-values()`, the input functions `doc()` and `collection()`. Two other frequently used functions are `not()` and `empty()`. The `not()` function is used in Boolean conditions; for instance, the following returns books where no author's last name is Stevens:

```
for $b in doc("books.xml")//book
where not(some $a in $b/author satisfies $a/last="Stevens")
return $b
```

The `empty()` function reports whether a sequence is empty. For instance, the following query returns books that have authors, but does not return the one book that has only editors:

```
for $b in doc("books.xml")//book
where not(empty($b/author))
return $b
```

The opposite of `empty()` is `exists()`, which reports whether a sequence contains at least one item. The preceding query could also be written as follows:

```
for $b in doc("books.xml")//book
where exists($b/author)
return $b
```

XQuery also has functions that access various kinds of information associated with a node. The most common accessor functions are `string()`, which returns the string value of a node, and `data()`, which returns the typed value of a node. These functions require some explanation. The string value of a node includes the string representation of the text found in the node and its descendants, concatenated in document order. For instance, consider the following query:

```
string((doc("books.xml")//author)[1])
```

The result of this query is the string `"Stevens  W."` (The exact result depends on the whitespace found in the original document—we have made some assumptions about what whitespace is present.)

User-Defined Functions

When a query becomes large and complex, it is often much easier to understand if it is divided into functions, and these functions can be reused in other parts of the query. For instance, we have seen a query that inverts a hierarchy to create a list of books by each author in a bibliography. It contained the following code:

```
for $b in doc("books.xml")/bib/book
where some $ba in $b/author satisfies
        ($ba/last=$l and $ba/first=$f)
order by $b/title
return $b/title
```

This code returns the titles of books written by a given author whose first name is bound to `$f` and whose last name is bound to `$l`. But you have to read all of the code in the query to understand that. Placing it in a named function makes its purpose clearer:

```
define function books-by-author($last, $first)
  as element()*
{
  for $b in doc("books.xml")/bib/book
  where some $ba in $b/author satisfies
         ($ba/last=$last and $ba/first=$first)
  order by $b/title
  return $b/title
}
```

XQuery allows functions to be recursive, which is often important for processing the recursive structure of XML. One common reason for using **recursive functions** is that XML allows recursive structures. For instance, suppose a book chapter may consist of sections, which may be nested. The query in Listing 1.19 creates a table of contents, containing only the sections and the titles, and reflecting the structure of the original document in the table of contents.

Listing 1.19 Query to Create a Table of Contents

```
define function toc($book-or-section as element())
  as element()*
{
  for $section in $book-or-section/section
  return
    <section>
      { $section/@* , $section/title , toc($section) }
    </section>
}

<toc>
   {
     for $s in doc("xquery-book.xml")/book
     return toc($s)
   }
</toc>
```

If two functions call each other, they are mutually recursive. Mutually recursive functions are allowed in XQuery.

Variable Definitions

A query can define a variable in the prolog. Such a variable is available at any point after it is declared. For instance, if access to the titles of books is used several times in a query, it can be provided in a variable definition:

```
define variable $titles { doc(“books.xml”)//title }
```

To avoid circular references, a variable definition may not call functions that are defined prior to the variable definition.

Library Modules

Functions can be put in library modules, which can be imported by any query. Every module in XQuery is either a main module, which contains a query body to be evaluated, or a library module, which has a module declaration but no query body. A library module begins with a module declaration, which provides a URI that identifies the module for imports, as shown in Listing 1.20.

Listing 1.20 Module Declaration for a Library Module

```
module "http://example.com/xquery/library/book"

define function toc($book-or-section as element())
  as element()*
{
  for $section in $book-or-section/section
  return
    <section>
      { $section/@* , $section/title , toc($section) }
    </section>
}
```

Functions and variable definitions in library modules are namespace-qualified. Any module can import another module using a module import, which specifies the URI of the module to be imported. It may also specify the location where the module can be found:

```
import module "http://example.com/xquery/library/book"
      at "file:///c:/xquery/lib/book.xq"
```

The location is not required in an import, since some implementations can locate modules without it. Implementations are free to ignore the location if they have another way to find modules.

A **namespace prefix** can be assigned in a module import, which is convenient since the functions in a module can only be called if a prefix has been assigned. The following query imports a module, assigns a prefix, and calls the function:

```
import module namespace b = "http://example.com/xquery/library/book"
      at "file:///c:/xquery/lib/book.xq"

<toc>
 {
   for $s in doc("xquery-book.xml")/book
   return b:toc($s)
 }
</toc>
```

When a module is imported, both its functions and its variables are made available to the importing module.

External Functions and Variables

XQuery implementations are often embedded in an environment such as a Java or C# program or a relational database. The environment can provide external functions and variables to XQuery. To access these, a query must declare them in the prolog:

```
define function outtie($v as xs:integer) as xs:integer external

define variable $v as xs:integer external
```

XQuery does not specify how such functions and variables are made available by the external environment, or how function parameters and arguments are converted between the external environment and XQuery.

Types in XQuery

Up to now, we have not spent much time discussing types, but the **type system** of XQuery is one of the most eclectic, unusual, and useful aspects of the language. XML documents contain a wide range of type information, from very loosely typed information without even a DTD, to rigidly structured data corresponding to relational data or objects. A language designed for processing XML must be able to deal with this fact gracefully; it must avoid imposing assumptions on what is allowed that conflict with what is actually found in the data, allow data to be managed without forcing the programmer to cast values frequently, and allow the programmer to focus on the documents being processed and the task to be performed rather than the quirks of the type system.

Consider the range of XML documents that XQuery must be able to process gracefully:

- XML may be strongly typed and governed by a W3C XML Schema, and a strongly typed query language with static typing can prevent many errors for this kind of data.
- XML may be governed by another schema language, such as DTDs or RELAX-NG.
- XML may have an *ad hoc* structure and no schema, and the whole reason for performing a query may be to discover the structure found in a document. For this kind of data, the query language should be able to process whatever data exists, with no preconceived notions of what should be there.
- XML may be used as a view of another system, such as a relational database. These systems are typically strongly typed, but do not use W3C XML Schema as the basis for their type system. Fortunately, standard mappings are emerging for some systems, such as SQL's mappings from relational schemas to W3C XML Schema. These are defined in the **SQL/XML** proposal, which provides standard XML extensions to SQL [SQLXML].
- XML data sources may have very complex structure, and expressions in XQuery must be well defined in terms of all the structures to which the operands can evaluate.

To meet these requirements, XQuery allows programmers to write queries that rely on very little type information, that take advantage of type information at run-time, or that take advantage of type information to detect potential errors before a query is ever executed. Chapter 4 provides a tutorial-like look at the topic of static typing in XQuery. Chapter 2, "Influences on the Design of XQuery," looks at the intricacies of some of the typing-related issues that members of the Work Group had to resolve.

Introduction to XQuery Types

The type system of XQuery is based on [SCHEMA]. There are two sets of types in XQuery: the built-in types that are available in any query, and types imported into a query from a specific schema. We will illustrate this with a series of functions that use increasing amounts of type information. XQuery specifies a conformance level called **Basic XQuery,** which is required for all implementations and allows two extensions: the schema import feature allows a W3C XML Schema to be imported in order to make its definitions available to the query, and the static typing feature allows a query to be compared to the imported schemas in order to catch errors without needing to access data. We will start with uses of types that are compatible with Basic XQuery. As we explore functions that require more type information, we will point out the points at which schema import and static typing are needed.

The first function returns a sequence of items in reverse order. The function definition does not specify the type of the parameter or the return type, which means that they may be any sequence of items:

```
define function reverse($items)
{
   let $count := count($items)
   for $i in 0 to $count
   return $items[$count - $i]
}
reverse( 1 to 5)
```

This function uses the `to` operator, which generates sequences of integers. For instance, the expression `1 to 5` generates the sequence 1, 2, 3, 4, 5. The `reverse` function takes this sequence and returns the sequence 5, 4, 3, 2, 1. Because this function does not specify a particular type for its parameter or return, it could also be used to return a sequence of some other type, such as a sequence of elements. Specifying more type information would make this function less useful.

Some functions take advantage of the known structures in XML or the built-in types of W3C XML Schema but need no advanced knowledge of any particular schema. The following function tests an element to see if it is the top-level element found in a document. If it is, then its parent node will be the document node, and the expression `$e/.. instance of document` will be true when evaluated for that node. The parameter type is `element`, since this is only defined for elements, and the return type is `xs:boolean`, which is a predefined type in XQuery and is the type of Boolean values:

```
define function is-document-element($e as element())
  as xs:boolean
{
  if ($e/.. instance of document-node())
    then true()
    else false()
}
```

All the built-in XML Schema types are predefined in XQuery, and these can be used to write **function signatures** similar to those found in conventional programming languages. For instance, the query in Listing 1.21 defines a function that computes the *n*th Fibonacci number and calls that function to create the first ten values of the Fibonacci sequence.

Listing 1.21 Query to Create the First Ten Fibonacci Numbers

```
define function fibo($n as xs:integer)
{
 if ($n = 0)
 then 0
 else if ($n = 1)
 then 1
 else (fibo($n - 1) + fibo($n - 2))
}

let $seq := 1 to 10
for $n in $seq
return <fibo n="{$n}">{ fibo($n) }</fibo>
```

Listing 1.22 shows the output of that query.

Listing 1.22 Results of the Query in Listing 1.21

```
<fibo n = "1">1</fibo>
<fibo n = "2">1</fibo>
<fibo n = "3">2</fibo>
<fibo n = "4">3</fibo>
<fibo n = "5">5</fibo>
<fibo n = "6">8</fibo>
<fibo n = "7">13</fibo>
<fibo n = "8">21</fibo>
<fibo n = "9">34</fibo>
<fibo n = "10">55</fibo>
```

Schemas and Types

On several occasions, we have mentioned that XQuery can work with untyped data, strongly typed data, or mixtures of the two. If a document is governed by a DTD or has no schema at all, then documents contain very little type information, and queries rely on a set of rules to infer an appropriate type when they encounter values at run-time. For instance, the following query computes the average price of a book in our bibliography data:

```
avg( doc("books.xml")/bib/book/price )
```

Since the bibliography does not have a schema, each price element is untyped. The `avg()` function requires a numeric argument, so it converts each price to a double and then computes the average. The conversion rules are discussed in detail in a later section. The implicit conversion is useful when dealing with untyped data, but prices are generally best represented as decimals rather than floating-point numbers. Later in this chapter we will present a schema for the bibliography in order to add appropriate type information. The schema declares price to be a decimal, so the average would be computed using decimal numbers.

Queries do not need to import schemas to be able to use built-in types found in data—if a document contains built-in types, the data model preserves type information and allows queries to access it. If we use the same query we used before to compute the average price, it will now compute the price as a decimal. This means that even Basic XQuery implementations, which are not able to import a schema, are able to use simple types found in the data. However, if a query uses logic that is related to the meaning of a schema, it is generally best to import the schema. This can only be done if an implementation supports the schema import feature. Consider the following function, which is similar to one discussed earlier:

```
define function books-by-author($author)
{
  for $b in doc("books.xml")/bib/book
  where some $ba in $b/author satisfies
         ($ba/last=$author/last and $ba/first=$author/first)
  order by $b/title
  return $b/title
}
```

Because this function does not specify what kind of element the parameter should be, it can be called with any element at all. For instance, a `book` element could be passed to this function. Worse yet, the query would not return an error, but would simply search for books containing an `author` element that exactly matches the book. Since such a match never occurs, this function always returns the empty sequence if called with a `book` element.

If an XQuery implementation supports the schema import feature, we can ensure that an attempt to call this function with anything but an `author` element would raise a type error. Let's assume that the namespace of this schema is `"urn:examples:xmp:bib"`. We can import this schema into a query and then use the element and attribute declarations and type definitions of the schema in our query, as shown in Listing 1.23.

Listing 1.23 Schema Import and Type Checking

```
import schema "urn:examples:xmp:bib" at "c:/dev/schemas/eg/bib.xsd"
default element namespace = "urn:examples:xmp:bib"
define function books-by-author($a as element(b:author))
  as element(b:title)*
{
  for $b in doc("books.xml")/bib/book
  where some $ba in $b/author satisfies
         ($ba/last=$a/last and $ba/first=$a/first)
  order by $b/title
  return $b/title
}
```

In XQuery, a type error is raised when the type of an expression does not match the type required by the context in which it appears. For instance, given the previous function definition, the function call in the following expression raises a type error, since an element named `book` can never be a valid author element:

```
for $b in doc("books.xml")/bib/book
return books-by-author($b)
```

All XQuery implementations are required to detect type errors, but some implementations detect them before a query is executed, and others detect them at run-time when query expressions are evaluated. The process of analyzing a query for type errors before a query is executed is called static typing, and it can be done using only the imported schema information and the query itself—there is no need for data to do static typing. In XQuery, static typing is an optional feature, but an implementation that supports static typing must always detect type errors statically, before a query is executed.

The previous example sets the default namespace for elements to the namespace defined by the schema. This allows the function to be written without namespace prefixes for the names in the paths. Another way to write this query is to assign a namespace prefix as part of the import and use it explicitly for element names. The query in Listing 1.24 is equivalent to the previous one.

Listing 1.24 Assigning a Namespace Prefix in Schema Imports

```
import schema namespace b = "urn:examples:xmp:bib"
  at "c:/dev/schemas/eg/bib.xsd"

define function books-by-author($a as element(b:author))
  as element(b:title)*
{
  for $b in doc("books.xml")/b:bib/b:book
  where some $ba in $b/b:author satisfies
         ($ba/b:last=$l and $ba/b:first=$f)
  order by $b/b:title
  return $b/b:title
}
```

When an element is created, it is immediately validated if there is a schema definition for its name. For instance, the following query raises an error because the schema definition says that a book must have a price:

```
import schema "urn:examples:xmp:bib" at "c:/dev/schemas/eg/bib.xsd"
default element namespace = "urn:examples:xmp:bib"

<book year="1994">
  <title>Catamaran Racing from Start to Finish</title>
  <author><last>Berman</last><first>Phil</first></author>
  <publisher>W.W. Norton & Company</publisher>
</book>
```

The schema import feature reduces errors by allowing queries to specify type information, but these errors are not caught until data with the wrong type information is actually encountered when executing a query. A query processor that implements the static typing feature can detect some kinds of errors by comparing a query to the imported schemas, which means that no data is required to find these errors. Let's modify our query somewhat and introduce a spelling error—`$a/first` is misspelled as `$a/firt` in Listing 1.25.

Listing 1.25 Query with a Spelling Error

```
import schema "urn:examples:xmp:bib" at "c:/dev/schemas/eg/bib.xsd"
default element namespace = "urn:examples:xmp:bib"

define function books-by-author($a as element(author))
  as element(title)*
{
  for $b in doc("books.xml")/bib/book
  where some $ba in $b/author satisfies
           ($ba/last=$a/last and $ba/first=$a/firt)
  order by $b/title
  return $b/title
}
```

An XQuery implementation that supports static typing can detect this error, because it has the definition for an author element, the function parameter is identified as such, and the schema says that an author element does not have a `firt` element. In an implementation that has schema import but not static typing, this function would actually have to call the function before the error would be raised.

However, in the following path expression, only the names of elements are stated:

```
doc("books.xml")/bib/book
```

XQuery allows element tests and attribute tests, node tests that are similar to the type declaration used for function parameters. In a path expression, the node test `element(book)` finds only elements with the same type as the globally declared `book` element, which must be found in the schemas that have been imported into the query. By using this instead of the name test `book` in the path expression, we can tell the query processor the element definition that will be associated with `$b`, which means that the **static type** system can guarantee us that a `$b` will contain title elements; see Listing 1.26.

Listing 1.26 Type Tests in Path Expressions

```
import schema "urn:examples:xmp:bib" at "c:/dev/schemas/eg/bib.xsd"
default element namespace = "urn:examples:xmp:bib"

define function books-by-author($a as element(author))
  as element(title)*
{
  for $b in doc("books.xml")/bib/element(book)
  where some $ba in $b/author satisfies
           ($ba/last=$a/last and $ba/first=$a/first)
  order by $b/title
  return $b/title
}
```

Sequence Types

The preceding examples include several queries in which the names of types use a notation that can describe the types that arise in XML documents. Now we need to learn that syntax in some detail. Values in XQuery, in general, are sequences, so the types used to describe them are called **sequence types.** Some types are built in and may be used in any query without importing a schema into the query. Other types are defined in W3C XML Schemas and must be imported into a query before they can be used.

Built-in Types

If a query has not imported a W3C XML Schema, it still understands the structure of XML documents, including types like document, element, attribute, node, text node, processing instruction, comment, ID, IDREF, IDREFS, etc. In addition to these, it understands the built-in W3C XML Schema simple types.

Table 1.4 lists the built-in types that can be used as sequence types.

In the notation for sequence types, **occurrence indicators** may be used to indicate the number of items in a sequence. The character `?` indicates zero or one items, `*` indicates zero or more items, and `+` indicates one or more items. Here are some examples of sequence types with occurrence indicators:

```
element()+                         One or more elements
xs:integer?                        Zero or one integers
document-node()*                   Zero or more document nodes
```

TABLE 1.4 Built-in Types That Can Be Used as Sequence Types

Sequence Type Declaration	What It Matches
`element()`	Any element node
`attribute()`	Any attribute node
`document-node()`	Any document node
`node()`	Any node
`text()`	Any text node
`processing-instruction()`	Any processing instruction node
`processing-instruction("xml-stylesheet")`	Any processing instruction node whose target is `xml-stylesheet`
`comment()`	Any comment node
`empty()`	An empty sequence
`item()`	Any node or atomic value
QName	An instance of a specific XML Schema built-in type, identified by the name of the type; e.g., `xs:string, xs:boolean, xs:decimal, xs:float, xs:double, xs:anyType, xs:anySimpleType`

When mapping XML documents to the XQuery data model, any element that is not explicitly given a simple or complex type by **schema validation** has the type `xs:anyType`. Any attribute that is not explicitly given a simple or complex type by schema validation has the type `xdt:untypedAtomic`. If a document uses simple or complex types assigned by W3C XML Schema, these are preserved in the data model.

Types from Imported Schemas

Importing a schema makes its types available to the query, including the definitions of elements and attributes and the declarations of complex types and simple types. We now present a schema for bibliographies, defining types that can be leveraged in the queries we use in the rest of this chapter. To support some of the examples, we have added an attribute that contains the ISBN number for each book, and have moved the publication year to an element. Listing 1.27 shows this schema—its relevant portions are explained carefully later in this section.

Listing 1.27 An Imported Schema for Bibliographies

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
     xmlns:bib="urn:examples:xmp:bib"
     targetNamespace="urn:examples:xmp:bib"
     elementFormDefault="qualified">

<xs:element name="bib">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="bib:book" minOccurs="0"
                     maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="book">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="title" type="xs:string"/>
      <xs:element ref="bib:creator" minOccurs="1"
              maxOccurs="unbounded"/>
      <xs:element name="publisher" type="xs:string"/>
      <xs:element name="price" type="currency"/>
      <xs:element name="year" type="xs:gYear"/>
    </xs:sequence>
    <xs:attribute name="isbn" type="bib:isbn"/>
  </xs:complexType>
</xs:element>
```

Listing 1.27 An Imported Schema for Bibliographies *(continued)*

```
<xs:element name="creator" type="person" abstract="true" />
<xs:element name="author" type="person"
substitutionGroup="bib:creator"/>
<xs:element name="editor" type="personWithAffiliation"
substitutionGroup="bib:creator"/>

<xs:complexType name="person">
  <xs:sequence>
    <xs:element name="last" type="xs:string"/>
    <xs:element name="first" type="xs:string"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="personWithAffiliation">
  <xs:complexContent>
    <xs:extension base="person">
     <xs:sequence>
       <xs:element name="affiliation" type="xs:string"/>
     </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:simpleType name="isbn">
  <xs:restriction base="xs:string">
    <xs:pattern value="[0-9]{9}[0-9X]"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="currency">
  <xs:restriction base="xs:decimal">
     <xs:pattern value="\d+.\d{2}"/>
  </xs:restriction>
</xs:simpleType>

</xs:schema>
```

Here is an example of a bibliography element that conforms to this new definition:

```
<bib xmlns="urn:examples:xmp:bib">
  <book isbn="0201563177">
    <title>Advanced Programming in the Unix Environment</title>
    <author><last>Stevens</last><first>W.</first></author>
    <publisher>Addison-Wesley</publisher>
    <price>65.95</price>
    <year>1992</year>
  </book>
</bib>
```

We do not teach the basics of XML Schema here—those who do not know XML Schema should look at XML Schema primer [SCHEMA]. However, to understand how XQuery leverages the type information found in a schema, we need to know what the schema says. Here are some aspects of the previous schema that affect the behavior of examples used in the rest of this chapter:

- All elements and types in this schema are in the namespace `urn:examples:xmp:bib` (for local elements, this was accomplished by using the `elementFormDefault` attribute at the top level of the schema). All attributes are in the null namespace.
- The following declaration says that the `isbn` type is a user-defined type derived from the `string` type by restriction and consists of nine digits followed either by a digit or by the character x:

```
<xs:simpleType name="isbn">
  <xs:restriction base="xs:string">
    <xs:pattern value="[0-9]{9}[0-9X]"/>
  </xs:restriction>
</xs:simpleType>
```

- The following declaration says that the "currency" type is derived from the decimal type by restriction, and must contain two places past the decimal point:

```
<xs:simpleType name="currency">
  <xs:restriction base="xs:decimal">
     <xs:pattern value="\d+.\d{2}"/>
  </xs:restriction>
</xs:simpleType>
```

- The following declarations say that `creator` is an abstract element that can never actually be created, and the `author` and `editor` elements are in the **substitution group** of `creator`:

```
<xs:element name="creator" type="person" abstract="true" />
<xs:element name="author" type="person"
substitutionGroup="bib:creator"/>
<xs:element name="editor" type="personWithAffiliation"
substitutionGroup="bib:creator"/>
```

- The content model for a book specifies a creator, but since creator is an abstract element, it can never be created—it will always match an author or an editor; see Listing 1.28.

Listing 1.28 Content Model for the Book Element

```
<xs:element name="book">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="title" type="xs:string"/>
      <xs:element ref="bib:creator" minOccurs="1"
           maxOccurs="unbounded"/>
      <xs:element name="publisher" type="xs:string"/>
      <xs:element name="price" type="currency"/>
      <xs:element name="year" type="xs:gYear"/>
    </xs:sequence>
    <xs:attribute name="isbn" type="bib:isbn"/>
  </xs:complexType>
</xs:element>
```

- The following elements are globally declared: `bib`, `book`, `creator`, `author`, `editor`. The type of the `bib` and `book` elements is "anonymous," which means that the schema does not give these types explicit names.
- All of the named types in this schema are global; in fact, in XML Schema, all named types are global.

Now let us explore the sequence type notation used to refer to constructs imported from the above schema. The basic form of an element test has two parameters: the name of the element and the name of the type:

```
element(creator, person)
```

To match an element, both the name and the type must match. The name will match if the element's name is `creator` or in the substitution group of `creator`; thus, in the above schema, the names `author` and `editor` would also match. The type will match if it is `person` or any other type derived from `person` by extension or restriction; thus, in the above schema, `personWithAffiliation` would also match. The second parameter can be omitted; if it is, the type is taken from the schema definition. Because the schema declares the type of `creator` to be `person`, the following declaration matches the same elements as the previous declaration:

```
element(creator)
```

In XML Schema, element and attribute definitions may be local, available only within a specific element or type. A context path may be used to identify a locally declared element or attribute. For instance, the following declaration matches the locally declared `price` element, which is found in the globally declared `book` element:

```
element(book/price)
```

Although this form is generally used to match locally declared elements, it will match any element whose name is `price` and which has the same type as the `price` element found in the globally declared `book` element. A similar form is used to match elements or attributes in globally defined types:

```
element(type(person)/last)
```

The same forms can be used for attributes, except that (1) attributes never have substitution groups in XML Schema; (2) attributes are not nillable in XML Schema; and (3) the element name is preceded by the `@` symbol in the XQuery syntax. For instance, the following declaration matches attributes named `price` of type `currency`:

```
attribute(@price, currency)
```

The following declaration matches attributes named `isbn` of the type found for the corresponding attribute in the globally declared `book` element:

```
attribute(book/@isbn)
```

Table 1.5 summarizes the declarations made available by importing the schema shown in Listing 1.27.

A sequence type declaration containing a name that does not match either a built-in type or a type imported from a schema is illegal and always raises an error.

There are no nillable elements in the sample schema. To indicate that an element test will also match a nilled element, the type should be declared nillable:

```
element(n, person nillable)
```

TABLE 1.5 The Effect of Importing the XML Schema in Listing 1.27

Sequence Type Declaration	**What It Matches**
`element(creator, person)`	An element named `creator` of type `person`
`element(creator)`	Any element named creator of type `xs:string`—the type declared for `creator` in the schema.
`element(*, person)`	Any element of type `person`.
`element(book/price)`	An element named price of type `currency`—`the` type declared for `price` elements inside a book element.
`element(type(person)/last)`	An element named last of type `xs:string`—the type declared for `last` elements inside the `person` type.
`attribute(@price, currency)`	An attribute named `price` of type `currency`.
`attribute(book/@isbn)`	An attribute named `isbn` of type `isbn`—the type declared for `isbn` attributes in a `book` element.
`attribute(@*, currency)`	Any attribute of type `currency`.
`bib:currency`	A value of the user-defined type `currency"`

The above declaration would match either an `n` element of type `person` or an `n` `person` which is nilled, such as this one, which uses `xsi:nil`:

```
<n xsi:nil="true" />
```

Working with Types

This section introduces various language features that are closely related to types, including function signatures, casting functions, typed variables, the `instance of` operator, `typeswitch`, and `treat as`.

Function Signatures

Parameters in a function signature may be declared with a sequence type, and the return type of a function may also be declared. For instance, the following function returns the discounted price of a book:

```
import schema namespace bib="urn:examples:xmp:bib"
define function discount-price($b as element(bib:book))
  as xs:decimal
```

```
{
  0.80 * $b//bib:price
}
```

It might be called in a query as follows:

```
for $b in doc("books.xml")//bib:book
where $b/bib:title = "Data on the Web"
return
  <result>
    {
      $b/bib:title,
      <price>{ discount-price($b/bib:price) }</price>
    }
  </result>
```

In the preceding query, the `price` element passed to the function exactly matches the declared type of the parameter. XQuery also defines some conversion rules that are applied if the argument does not exactly match the type of the parameter. If the type of the argument does not match and cannot be converted, a type error is raised. One important conversion rule is that the value of an element can be extracted if the expected type is an atomic type and an element is encountered. This is known as atomization. For instance, consider the query in Listing 1.29.

Listing 1.29 Atomization

```
import schema namespace bib="urn:examples:xmp:bib"

define function discount-price($p as xs:decimal)
  as xs:decimal
{
  0.80 * $p//bib:price
}

for $b in doc("books.xml")//bib:book
where $b/bib:title = "Data on the Web"
return
  <result>
    {
      $b/bib:title,
      <price>{ discount-price($b/bib:price) }</price>
    }
  </result>
```

When the typed value of the `price` element is extracted, its type is `bib:currency`. The function parameter expects a value of type `xs:decimal`, but the schema imported into the query says that the currency type is derived from `xs:decimal`, so it is accepted as a decimal.

In general, the typed value of an element is a sequence. If any value in the argument sequence is untyped, XQuery attempts to convert it to the required type and raises a type error if it fails. For instance, we can call the revised `discount-price()` function as follows:

```
let $w := <foo>12.34</foo>
return discount-price($w)
```

In this example, the `foo` element is not validated, and contains no type information. When this element is passed to the function, which expects a decimal, the function first extracts the value, which is untyped. It then attempts to cast 12.34 to a decimal; because 12.34 is a legitimate lexical representation for a decimal, this cast succeeds. The last conversion rule for function parameters involves **type promotion**: If the parameter type is `xs:double`, an argument whose type is `xs:float` or `xs:decimal` will automatically be cast to the parameter type; if the parameter type is `xs:float`, an argument whose type is `xs:decimal` will automatically be cast to the parameter type.

The parameter type or the return type may be any sequence type declaration. For instance, we can rewrite our function to take a `price` element, which is a locally declared element, by using a context path in the sequence type declaration:

```
import schema namespace bib="urn:examples:xmp:bib"

define function discount-price($p as element(bib:book/bib:price))
  as xs:decimal
{
  0.80 * $p
}
```

If the `price` element had an **anonymous type,** this would be the only way to indicate a `price` element of that type. Since our schema says a `price` element has the type `bib:currency`, the preceding function is equivalent to this one:

```
import schema namespace bib="urn:examples:xmp:bib"

define function discount-price($p as element(bib:price, bib:currency))
```

```
  as xs:decimal
{
  0.80 * $p
}
```

The same conversion rules that are applied to function arguments are also applied to function return values. Consider the following function:

```
define function decimate($p as element(bib:price, bib:currency))
  as xs:decimal
{
    $p
}
```

In this function, `$p` is an element named `bib:price` of type `bib:currency`. When it is returned, the function applies the function conversion rules, extracting the value, which is an atomic value of type `bib:currency`, then returning it as a valid instance of `xs:decimal`, from which its type is derived.

Casting and Typed Value Construction

Casting and typed value construction are closely related in XQuery. Constructor functions can be used to do both. In XQuery, any built-in type is associated with a constructor function that is found in the XML Schema namespace and has the same name as the type it constructs. This is the only way to create some types, including most date types. Here is a constructor for a date:

```
xs:date("2000-01-01")
```

Constructor functions check a value to make sure that the argument is a legal value for the given type and raise an error if it is not. For instance, if the month had been `13`, the constructor would have raised an error.

Constructor functions are also used to cast values from one type to another. For instance, the following query converts an integer to a string:

```
xs:string( 12345 )
```

Some types can be cast to each other, others cannot. The set of casts that will succeed can be found in [XQ-FO]. Constructor functions are also created for imported simple types—this is discussed in the section on imported schemas.

When a schema is imported and that schema contains definitions for simple types, constructor functions are automatically created for these types. Like the built-in constructor functions, these functions have the same name as the type that is constructed. For instance, the `currency` type in our bibliography schema limits values to two digits past the decimal, and the `isbn` type restricts ISBN numbers to nine digits followed by either another digit or the letter `X`. Importing this schema creates constructor functions for these two types. The following expression creates an atomic value of type `isbn`:

```
import schema namespace bib="urn:examples:xmp:bib"
bib:isbn("012345678X")
```

The constructor functions for types check all the **facets** for those types. For instance, the following query raises an error because the pattern in the type declaration says that an ISBN number may not end with the character `Y`:

```
import schema namespace bib="urn:examples:xmp:bib"
bib:isbn("012345678Y")
```

Typed Variables

Whenever a variable is bound in XQuery, it can be given a type by using an `as` clause directly after the name of the variable. If a value that is bound to a typed variable does not match the declared type, a type error is raised. For instance, in the query shown in Listing 1.30, the `let` clause states that `$authors` must contain one or more `author` elements.

Listing 1.30 Declaring the Type of a Variable

```
import schema namespace bib="urn:examples:xmp:bib"

for $b in doc("books.xml")//bib:book
let $authors as element(bib:author)+ := $b//bib:author
return
  <result>
    {
      $b/bib:title,
      $authors
    }
</result>
```

Since the schema for a bibliography allows a book to have editors but no authors, this query will raise an error if such a book is encountered. If a programmer simply assumed all books have authors, using a typed variable might identify an error in a query.

The `instance of` *Operator*

The `instance of` operator tests an item for a given type. For instance, the following expression tests the variable `$a` to see if it is an `element` node:

```
$a instance of element()
```

As you recall, literals in XQuery have types. The following expressions each return `true`:

```
<foo/> instance of element()

3.14 instance of xs:decimal

"foo" instance of xs:string

(1, 2, 3) instance of xs:integer*

() instance of xs:integer?
(1, 2, 3) instance of xs:integer+
```

The following expressions each return false:

```
3.14 instance of xdt:untypedAtomic

"3.14" instance of xs:decimal

3.14 instance of xs:integer
```

Type comparisons take type hierarchies into account. For instance, recall that SKU is derived from `xs:string`. The following query returns `true`:

```
import schema namespace bib="urn:examples:xmp:bib"
bib:isbn("012345678X") instance of xs:string
```

The `typeswitch` *Expression*

The `typeswitch` expression chooses an expression to evaluate based on the dynamic type of an input value—it is similar to the `CASE` statement found in several programming languages, but it branches based on the argument's type, not on its value. For instance, suppose we want to write a function that creates a simple wrapper element around a value, using `xsi:type` to preserve the type of the wrapped element, as shown in Listing 1.31.

Listing 1.31 Function Using the `typeswitch` Expression

```
define function wrapper($x as xs:anySimpleType)
  as element()
{
typeswitch ($x)
      case $i as xs:integer
           return <wrap xsi:type="xs:integer">{ $i }</wrap>
      case $d as xs:decimal
           return <wrap xsi:type="xs:decimal">{ $d }</wrap>
      default
           return error("unknown type!")
}

wrapper( 1 )
```

The case clause tests to see if `$x` has a certain type; if it does, the case clause creates a variable of that type and evaluates the associated return clause. The `error` function is a standard XQuery function that raises an error and aborts execution of the query. Here is the output of the query in Listing 1.31:

```
<wrap xsi:type="xs:integer">1</wrap>
```

The case clauses test to see if `$x` has a certain type; if it does, the case clause creates a variable of that type and evaluates the first return clause that matches the type of `$x`. In this example, 1 is both an integer and a decimal, since `xs:integer` is derived from `xs:decimal` in XML Schema, so the first matching clause is evaluated. The error function is a standard XQuery function that raises an error and aborts execution of the query.

The `typeswitch` expression can be used to implement a primitive form of polymorphism. For instance, suppose authors and editors are paid different percentages of the total price of a book. We could write the function shown in Listing 1.32, which invokes the appropriate function to calculate the payment based on the substitution group hierarchy.

Listing 1.32 Using `typeswitch` to Implement Simple Polymorphism

```
import schema namespace bib="urn:examples:xmp:bib"

define function pay-creator(
    $c as element(bib:creator),
    $p as xs:decimal)
{
  typeswitch ($c)
      case $a as element(bib:author)
           return pay-author($a, $p)
      case $e as element(bib:editor)
           return pay-editor($e, $p)
      default
           return error("unknown creator element!")
}
```

The `treat as` *Expression*

The `treat as` expression asserts that a value has a particular type, and raises an error if it does not. It is similar to a cast, except that it does not change the type of its argument, it merely examines it. `Treat as` and `instance of` could be used together to write the function shown in Listing 1.33, which has the same functionality as the function in Listing 1.32.

Listing 1.33 Using `treat as` and `instance of` to Implement Simple Polymorphism

```
import schema namespace bib="urn:examples:xmp:bib"

define function pay-creator(
  $c as element(bib:creator),
  $p as xs:decimal)
{
if ($c instance of element(bib:author))
then pay-author($a, $p)
else if ($c instance of element(bib:editor))
then pay-editor($e, $p)
else error("unknown creator element!")
}
```

In general, `typeswitch` is preferable for this kind of code, and it also provides better type information for processors that do static typing.

Implicit Validation and Element Constructors

We have already discussed the fact that **validation** of the elements constructed in a query is automatic if the declaration of an element is global and is found in a schema that has been imported into the query. Elements that do not correspond to a global element definition are not validated. In other words, element construction uses XML Schema's **lax validation mode**. The query in Listing 1.34 creates a fully validated `book` element, with all the associated type information.

Listing 1.34 Query That Creates a Fully Validated Book Element

```
import schema namespace bib="urn:examples:xmp:bib"

<bib:book isbn="0201633469">
  <bib:title>TCP/IP Illustrated</bib:title>
  <bib:author>
    <bib:last>Stevens</bib:last>
    <bib:first>W.</bib:first>
  </bib:author>
  <bib:publisher>Addison-Wesley</bib:publisher>
  <bib:price>65.95</bib:price>
  <bib:year>1994</bib:year>
</bib:book>
```

Because element constructors validate implicitly, errors are caught early, and the types of elements may be used appropriately throughout the expressions of a query. If the element constructor in Listing 1.34 had omitted a required element or misspelled the name of an element, an error would be raised.

Relational programmers are used to writing queries that return tables with only some columns from the original tables that were queried. These tables often have the same names as the original tables, but a different structure. Thus, a relational programmer is likely to write a query like the following:

```
import schema namespace bib="urn:examples:xmp:bib"

for $b in doc("books.xml")//bib:book
return
```

```
  <bib:book>
    {
      $b/bib:title,
      $b//element(bib:creator)
    }
</bib:book>
```

This query raises an error, because the `bib:book` element that is returned has a structure that does not correspond to the schema definition. Validation can be turned off using a `validate` expression, as shown in Listing 1.35, which uses `skip`.

Listing 1.35 Using `validate` to Disable Validation

```
import schema namespace bib="urn:examples:xmp:bib"

for $b in doc("books.xml")//bib:book
return
 validate skip
 {
  <bib:book>
    {
      $b/bib:title,
      $b//element(bib:creator)
    }
   </bib:book>
}
```

The `validate` expression can also be used to specify a **validation context** for locally declared elements or attributes. For instance, the `price` element is locally declared:

```
import schema namespace bib="urn:examples:xmp:bib"

validate context bib:book
 {
  <bib:price>49.99</bib:price>
 }
```

If an element's name is not recognized, it is treated as an untyped element unless `xsi:type` is specified. For instance, the following query returns a well-formed element with untyped content, because the `bib:mug` element is not defined in the schema:

```
import schema namespace bib="urn:examples:xmp:bib"
<bib:mug>49.99</bib:mug>
```

A query can specify the type of an element using the `xsi:type` attribute; in this case, the element is validated using the specified type:

```
import schema namespace bib="urn:examples:xmp:bib"
<bib:mug xsi:type="xs:decimal">49.99</bib:mug>
```

If a locally declared element is not wrapped in a `validate` expression that specifies the context, it will generally be treated as a well-formed element with untyped content, as in the following query:

```
import schema namespace bib="urn:examples:xmp:bib"
<bib:price>49.99</bib:price>
```

To prevent errors like this, you can set the default validation mode to `strict`, which means that all elements must be defined in an imported schema, or an error is raised. This is done in the prolog. The following query raises an error because the `bib:price` element is not recognized in the global context:

```
import schema namespace bib="urn:examples:xmp:bib"
validation strict
<bib:price>49.99</bib:price>
```

The validation mode may be set to `lax`, which is the default behavior, `strict`, as shown above, or `skip` if no validation is to be performed in the query.

Summary

XQuery is not only a query language, but also a language that can do fairly general processing of XML. It is a strongly typed language that works well with data that may be strongly or weakly typed. Because the types used in XQuery are the same types used in XML and XML Schema, the type system is a better match for the data that is being processed. If the XML is governed only by a DTD or has no schema, the appropriate types are document, element, attribute, node, text node, processing instruction, comment, ID, IDREF, IDREFS, and so on. A strongly typed language that does not support these types tends to get in the way, because it is a poor match for the data being processed, and the language insists on the wrong things. If W3C XML Schema types are

Katz_C01.qxd 7/23/03 2:45 PM Page 78

present in the data, these types are observed as well. Implementations and users of XQuery can work at various levels of typing by deciding whether to import schemas, whether to use static typing, and whether to set the validation mode to `strict`, `lax`, or `skip`.

XQuery was designed to be compact and compositional, and to be well suited for views of data that is not physically stored as XML. Both data integration and general purpose XML processing are likely to be important applications of XQuery. In practice, queries written in XQuery tend to be well suited to the kinds of tasks for which XML is generally used.