CHAPTER 7

XML

XML (extensible Markup Language) is a data representation standard adopted by the World Wide Web Consortium (W3C) many years ago as the mechanism to share and exchange data on the Web. This chapter introduces the student to basic XML concepts including syntax, querying using XPath and XQuery, and schema specifications using XML Schema.

7.1 XML Basics

XML represents data in text format and encloses data items between user understandable and meaningful tags. For example, the address of a student is described as follows:

```
<address>123 Main Street, Atlanta, GA 30002</address>
```

The data item in this case is "`123 Main Street, Atlanta, GA 30002`" and is enclosed between the start tag `<address>` and the end tag `</address>`. The tag name "`address`" is user-defined and is descriptive of the data item that it is enclosing.

The entire string starting with the start tag, including the data item, and ending with the end tag is referred to as an XML *element*. XML elements can be nested as in the following example which includes sub-components of the address:

```
<address>
  <street>123 Main Street</street>
  <city>Atlanta</city>
  <state>GA</state>
  <zipcode>30002</zipcode>
</address>
```

Here, the `<address>` XML element has four sub-elements, `<street>`, `<city>`, `<state>`, and `<zipcode>`. Notice that all sub-elements are completely enclosed with the start and end tags of the main element.

Occasionally, there is a need to have an element without any contents. Such elements are called *empty* elements. For example, the following empty element may be used in a situation where the phone number is not known:

```
<phone></phone>
```

The above empty element can also be represented in the shorter notation:

```
<phone/>
```

There is no restriction in XML for repeating the sub-element tags. So, a list of phone numbers for an individual can be described as follows:

```
<person>
  <name>John Smith</name>
  <phone>111-1234</phone>
  <phone>212-2121</phone>
</person>
```

In addition to the element syntactic structure, XML also provides an additional mechanism to describe data. XML *attributes* are name-value pairs that are introduced in the start tag of elements. The following is an example of attributes in use within an element description:

```
<cost currency=”USD”>25.20</cost>
```

Here, the cost of some item is being described. The cost (25.20) is being described using an XML element <cost>. The start tag includes a name-value pair `currency=”USD”` indicating that the currency of the cost is in US dollars. In contrast to XML elements, attribute names cannot be repeated within the same start-tag. So, the following will be an error:

```
<cost currency=”USD” currency=”INR”>25.20</cost>
```

XML syntax also allows mixing of elements and text outside of the element context. This is a left over feature from the document world in mark up languages are still used. For example, consider the following XM fragment code:

```
<person>
  <name>John</name>
  This is my cousin!
  <age>22</age>
</person>
```

In this description of a person, the text “`This is my cousin!`” appears outside of the context of any element of sub-element. Such annotations are usually ignored by XML parsers.
Comments in XML are introduced as follows:

```
<!--This is a comment -->
```

XML documents usually begin with a version statement as follows:

```
<?xml version=”1.0”>
```

The `CDATA` construct allows one to include special characters such as the < or > symbols as part of text. For example, to include XML tags as part of the content of elements, the `CDATA` construct can be used as follows:

```
<![CDATA[<age>22</age>]]>
```

7.2 Company Database in XML

One possible XML representation of the COMPANY database is discussed in this section. The overall structure of the XML document is as follows:

```
<?xml version="1.0">
<companyDB>
  <departments>
  …
  …
  </departments>
  <employees>
  …
  …
  </employees>
  <projects>
  …
  …
  </projects>
</companyDB>
```

The document contains three main sections, one each for the list of departments, employees, and projects. Each section describes the individual entities within the classes along with relationships with other entities.

The `<departments>` element contains one or more `<department>` elements that describe the individual department along with its relationships with other entities. The following XML code fragment shows the details for department `5`:

```
<departments>
  …
  …
  <department dno="5">
    <dname>Research</dname>
    <locations>
      <location>Bellaire</location>
      <location>Sugarland</location>
      <location>Houston</location>
    </locations>
    <manager mssn="333445555">
      <startDate>22-MAY-1978</startDate>
    </manager>
    <employees essns="123456789 333445555 666884444 453453453"/>
    <projectsControlled pnos="1 2 3"/>
  </department>
  …
  …
</departments>
```

As can be seen, the department element contains an attribute `dno` and several sub-elements, each describing either a simple attribute or a relationship. The `<dname>` sub-element describes the department name, the `<locations>` sub-element encloses one or more `<location>` elements, each describing a department location. The `<manager>`, `<employees>`, and `<projectsControlled>` sub-elements describe relationships with other entities and all these are represented by XML attributes.

The `<employees>` element contains one or more `<employee>` elements that describe the individual employees along with its relationships with other entities. The following XML code fragment shows the details for employee `333445555`:

```
<employees>
  …
  …
  <employee ssn="333445555" worksFor="5"
            supervisor="888665555" manages="5">
    <fname>Franklin</fname>
    <minit>T</minit>
    <lname>Wong</lname>
    <dob>08-DEC-45</dob>
    <address>638 Voss, Houston, TX</address>
    <sex>M</sex>
    <salary>40000</salary>
    <dependents>
      <dependent>
        <dependentName>Alice</dependentName>
        <sex>F</sex>
        <dob>05-APR-1976</dob>
        <relationship>Daughter</relationship>
      </dependent>
      <dependent>
        <dependentName>Theodore</dependentName>
        <sex>M</sex>
        <dob>25-OCT-1973</dob>
        <relationship>Son</relationship>
      </dependent>
      <dependent>
        <dependentName>Joy</dependentName>
        <sex>F</sex>
        <dob>03-MAY-1948</dob>
        <relationship>Spouse</relationship>
      </dependent>
    </dependents>
    <supervisees essns="123456789 666884444 453453453"/>
    <projects>
      <worksOn pno="2" hours="10.0"/>
      <worksOn pno="3" hours="10.0"/>
      <worksOn pno="10" hours="10.0"/>
      <worksOn pno="20" hours="10.0"/>
    </projects>
  </employee>
  …
  …
</employees>
```

The `<projects>` element contains one or more `<project>` elements that describe the individual projects along with its relationships with other entities. The following XML code fragment shows the details for project `1`:

```
<projects>
  …
  …
  <project pnumber="1" controllingDepartment="5">
    <pname>ProductX</pname>
    <plocation>Bellaire</plocation>
    <workers>
      <worker essn="123456789">32.5</worker>
      <worker essn="453453453">20.0</worker>
    </workers>
  </project>
  …
  …
</projects>
```

We shall use the above XML description of the COMPANY database in subsequent sections to discuss XML querying.

7.3 XML Editor EditiX

A free version of an XML editor can be downloaded from http://free.editix.com/. This software is available for all platforms including PC, Mac, and Linux. The free version of the editix has built-in support for editing and validating XML documents against DTDs and XML Schemas and also provides for XPath and XQuery support. The initial window for editix is shown in Figure 7.1.

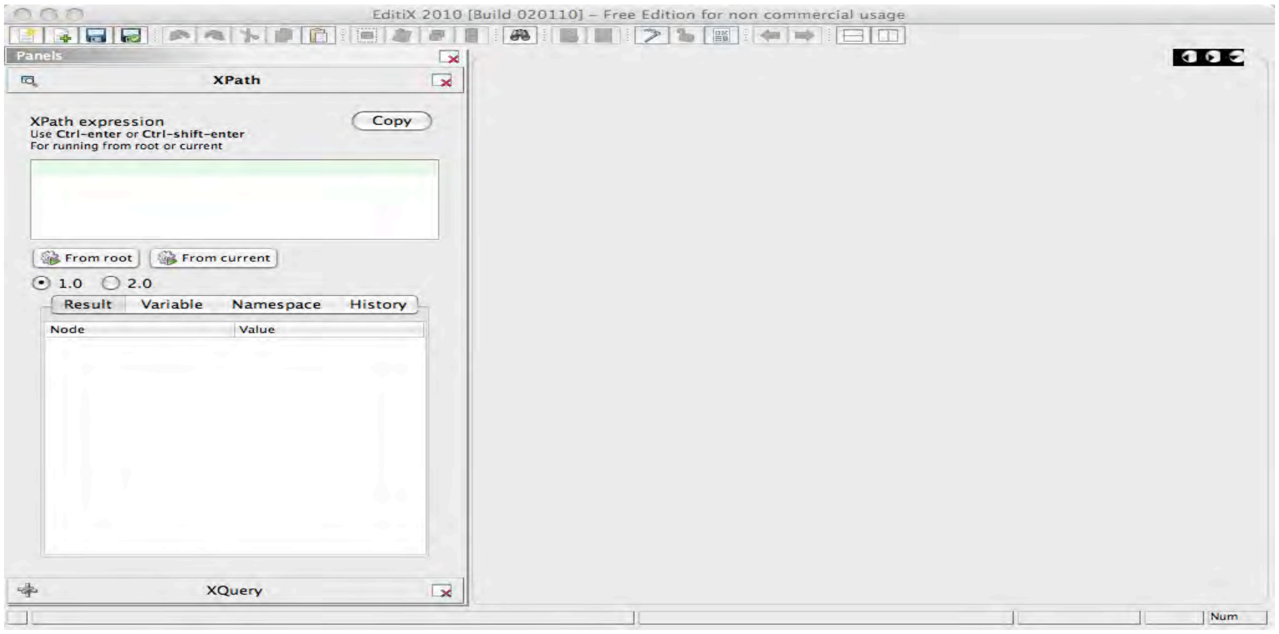
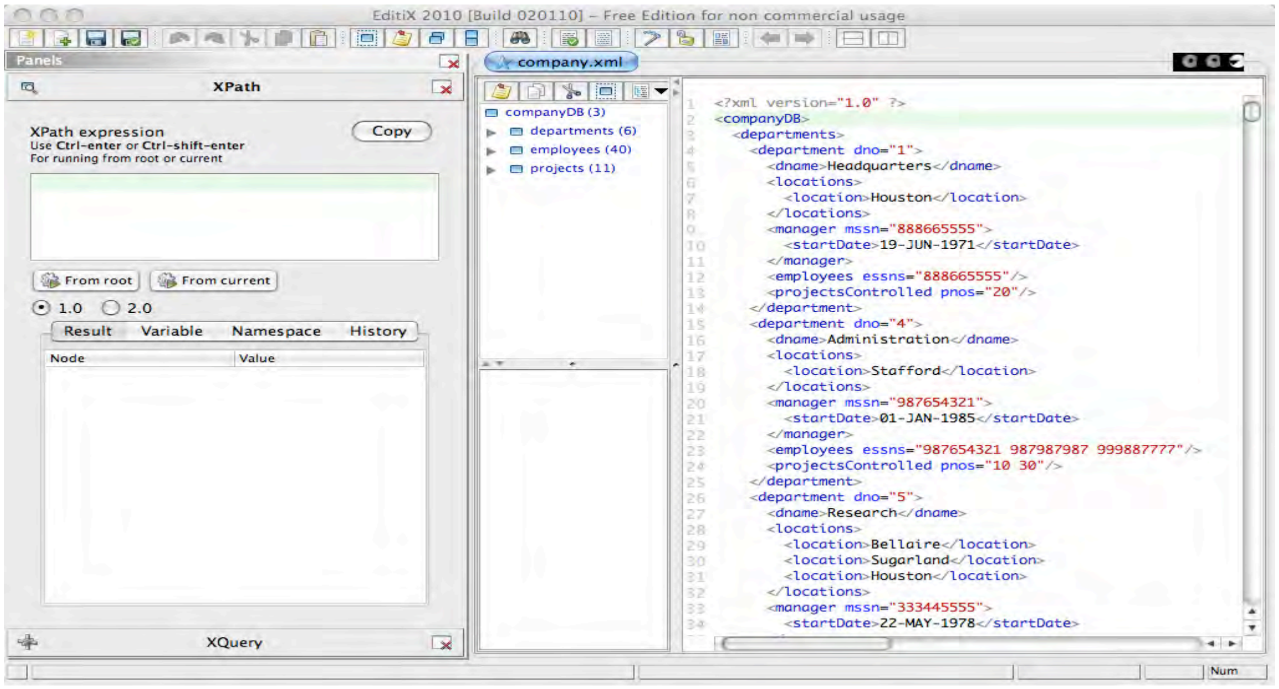

EditiX 2010 [Build 020110] – Free Edition for non commercial usage
Panels
XPath
XPath expression
Use Ctrl-enter or Ctrl-shift-enter
For running from root or current
Copy
From root
From current
1.0
2.0
Result
Variable
Namespace
History
Node
Value
XQuery
Num

Figure 7.1 editix Window on startup

The left panel provides space to specify XPath as well as XQuery expressions to be evaluated against an XML document. The right panel is initially blank. This space is used to display the XML document. The File menu provides the ability to assign DTD or XML Schema files to the XML document as well as to validate XML documents against the schema files. Figure 7.2 shows the window after the `company.xml` document is opened in editx.



```xml
<?xml version="1.0" ?>
<companyDB>
  <departments>
    <department dno="1">
      <dname>Headquarters</dname>
      <locations>
        <location>Houston</location>
      </locations>
      <manager mssn="888665555">
        <startDate>19-JUN-1971</startDate>
      </manager>
      <employees essns="888665555"/>
      <projectsControlled pnos="20"/>
    </department>
    <department dno="4">
      <dname>Administration</dname>
      <locations>
        <location>Stafford</location>
      </locations>
      <manager mssn="987654321">
        <startDate>01-JAN-1985</startDate>
      </manager>
      <employees essns="987654321 987987987 999887777"/>
      <projectsControlled pnos="10 30"/>
    </department>
    <department dno="5">
      <dname>Research</dname>
      <locations>
        <location>Bellaire</location>
        <location>Sugarland</location>
        <location>Houston</location>
      </locations>
      <manager mssn="333445555">
        <startDate>22-MAY-1978</startDate>
```

Figure 7.2 editix Window after opening XML document

The right panel contains two parts: a navigation pane that shows top-level elements along with counts of sub-elements and the display pane showing the entire XML document. The navigation pane can be used to go to specific portions of the XML file. The editor itself is quite straightforward to use.

7.4 XPath

XPath is an important specification language used to traverse and locate nodes in an XML document. It is not a full-fledged query language, but is used to specify a set of nodes in the XML tree structure, very much like the file specification in the directory hierarchy of a Unix operating system. XPath expressions are used in other query languages such as XQuery. The `company.xml` document will be used to illustrate all examples in the rest of the chapter.

There are several types of nodes in the XML tree: root node (e.g. `<companyDB>`), element node (e.g. `<salary>80000</salary>`), attribute node (e.g. `dno="4"`), and text node (e.g. `Stafford`). The nodes are related to each other in the XML tree via the parent, child, sibling, ancestor, and descendant relationships.

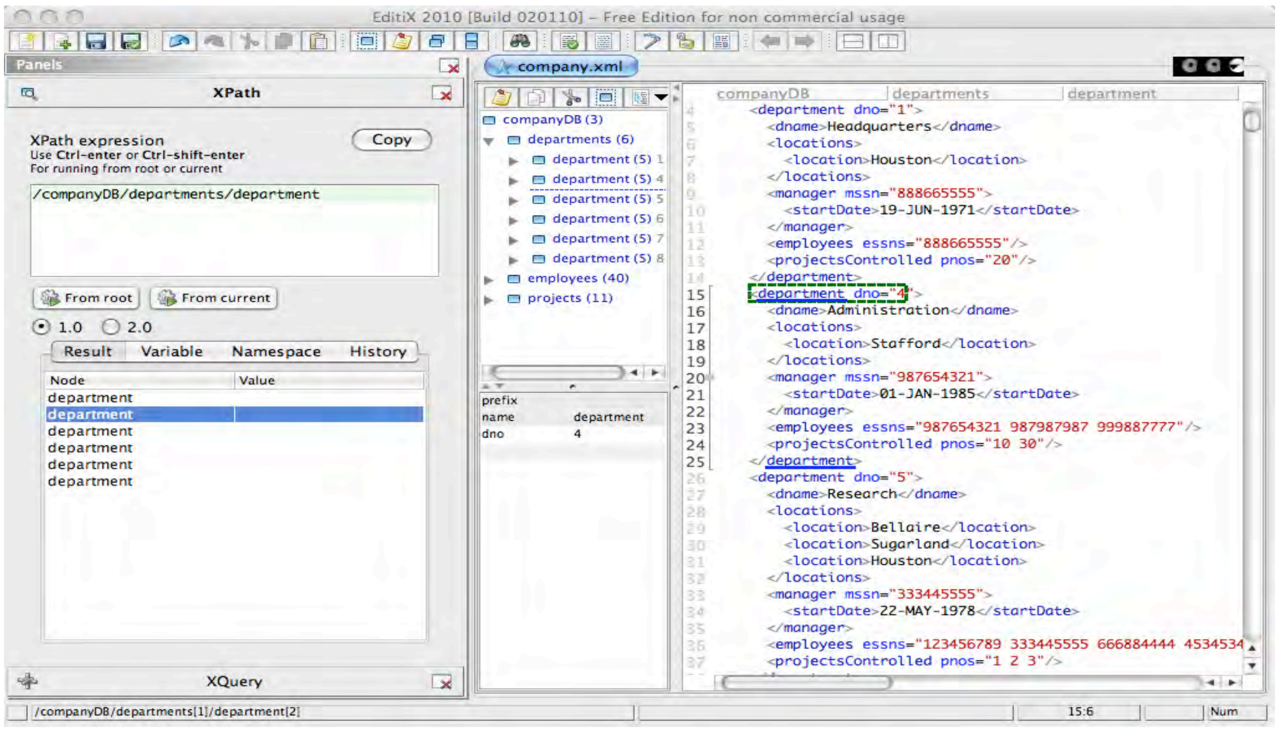
The latest XPath specification is available from the Web site http://www.w3.org/TR/xpath and the list of built-in functions for use in XPath and XQuery is available at http://www.w3.org/TR/xquery-operators/.

Basic XPath expressions

XPath expressions are of two types: absolute and relative. An absolute XPath expression begins with a / and is followed by a sequence of XML element names each separated by a /. For example, the expression:

```
/companyDB/departments/department
```

denotes the set of all `<department>` elements that are sub-elements of `<departments>` elements which in turn are sub-elements of the root element `<companyDB>` in the XML document. To execute XPath expressions on editix, simply enter the XPath expression in the XPath panel on the left and click "From root" button. This will bring up the matching nodes in the results box below. Upon clicking one of the results, the corresponding section in the XML tree display on the right is highlighted. The window after executing the above XPath expression is shown in Figure 7.3.



XPath expression
Use Ctrl-enter or Ctrl-shift-enter
For running from root or current

```
/companyDB/departments/department
```

[✓ 1.0] 2.0

Node	Value
department	
department	
department	
department	
department	
department	

```
companyDB (3)
  departments (6)
    department (5) 1
    department (5) 4
    department (5) 5
    department (5) 6
    department (5) 7
    department (5) 8
  employees (40)
  projects (11)
```

prefix	
name	department
dno	4

```
<department dno="1">
  <dname>Headquarters</dname>
  <locations>
    <location>Houston</location>
  </locations>
  <manager mssn="888665555">
    <startDate>19-JUN-1971</startDate>
  </manager>
  <employees essns="888665555"/>
  <projectsControlled pnos="20"/>
</department>
<department dno="4">
  <dname>Administration</dname>
  <locations>
    <location>Stafford</location>
  </locations>
  <manager mssn="987654321">
    <startDate>01-JAN-1985</startDate>
  </manager>
  <employees essns="987654321 987987987 999887777"/>
  <projectsControlled pnos="10 30"/>
</department>
<department dno="5">
  <dname>Research</dname>
  <locations>
    <location>Bellaire</location>
    <location>Sugarland</location>
    <location>Houston</location>
  </locations>
  <manager mssn="333445555">
    <startDate>22-MAY-1978</startDate>
  </manager>
  <employees essns="123456789 333445555 666884444 4534534
  <projectsControlled pnos="1 2 3"/>
```

```
/companyDB/departments[1]/department[2]        15:6        Num
```

Figure 7.3 editix Window after XPath expression is executed

The expression / denotes the root node, . denotes the current node, and .. denotes the parent node of the current node. Expressions that do not begin with the / symbol are called relative XPath expressions and have a meaning only with respect to a current node. Attribute values are accessed using the @ expression. For example, the following XPath expression can be used to access the supervisor attribute of employee elements in the XML document:

```
/companyDB/employees/employee/@supervisor
```

Text nodes in the XML document can be accessed using the built-in function `text()`. For example, to access all the `lname` values for employees, the following XPath expression can be used:

```
/companyDB/employees/employee/lname/text()
```

Advanced XPath expressions

XPath allows the * wildcard to match any node in the path. For example, to select all children nodes of `<employee>` element, the following XPath expression can be used:

```
/companyDB/employees/employee/*
```

The wildcard can also be used to retrieve all attributes of an element. For example, to retrieve all the attributes of the `<employee>` element, the following XPath expression can be used:

```
/companyDB/employees/employee/@*
```

Another wildcard provided by XPath is `//`, the descendant-or-self wildcard. This allows access to nodes at any level in the tree. For example, to access all the `<dob>` elements in the XML document regardless of the level it appears, the following XPath expressions can be used:

```
//dob
```

XPath allows access to the children nodes based on their positions. For example, the 7th <employee> sub-element of <employees> element can be accessed using the XPath expression:

```
/companyDB/employees/employee[7]
```

To access the second dependent of the first employee, the following XPath expression can be used:

```
/companyDB/employees/employee[1]/dependents/dependent[2]
```

Using the built-in function last(), the last dependent of the first employee can be accessed using the XPath expression:

```
/companyDB/employees/employee[1]/dependents/dependent[last()]
```

The [] notation used to get positional access to the children of a node can be used to express general predicates as well. A logical predicate involving the attributes and child node values of the current node can be specified along with any built-in functions. Several examples of the use of general predicate are presented next.

The `position()` built-in function can be used within the predicate to access the first three `<employee>` elements in the XML document as follows:

```
/companyDB/employees/employee[position()<=3]
```

To access all `<employee>` elements that have a `<minit>` sub-element with a value of "`E`", the following XPath expression can be used:

```
/companyDB/employees/employee[minit="E"]
```

To access all `<project>` elements that have a `controllingDepartment` attribute value greater than `6`, the following XPath expression can be used:

```
/companyDB/projects/project[@controllingDepartment>6]
```

The `starts-with()` and `contains()` built-in functions work on string values and can be useful to access elements based on substring matches. To access all `<employee>` elements whose last name starts with "`S`", the following XPath expression can be used:

```
/companyDB/employees/employee[starts-with(lname,"S")]
```

To access all `<employee>` elements who address contains the sub-string "`Philadelphia`", the following XPath expression can be used:

```
/companyDB/employees/employee[contains(address,"Philadelphia")]
```

Complex search conditions can be specified using logical connectives "`and`", "`or`" and "`not`". For example, to access all `<employee>` elements who work for department `7` and who are males and who have a male dependent, the following XPath expression can be used:

```
//employee[@worksFor=7 and sex="M" and dependents/dependent[sex="M"]]
```

XPath allows a path expression to be used in place of a predicate with the [] notation. The node is selected if the expression within the square brackets evaluates to a non-empty set if nodes. For example, the XPath expression:

```
/companyDB/employees/employee[dependents]
```

returns all employee nodes that have dependents.

XPath also provides support for the "union" of two sub-expressions using the "|" operator. For example, to access all project names and department names, the following XPath expression can be used:

```
/companyDB/departments/department/dname/text() |
/companyDB/projects/project/pname/text()
```

The general form of an XPath expression is one of the following:

`/locationStep1/locationStep2/…` for absolute path expressions

or

`locationStep1/locationStep2/…` for relative path expressions.

In either case, the location step is of the form:

```
axis::nodeSelector[selectionPredicate]
```

where axis is one of the following: `child`, `parent`, `descendant`, `ancestor`, `descendant-or-self (//)`, or `ancestor-or-self`. The default axis is `child`. The `nodeSelector` is either the name of an element or an attribute or a wild card symbol. The `selectionPredicate` is a predicate as discussed in various examples previously. An example of an XPath expression in which the axis is explicitly mentioned is:

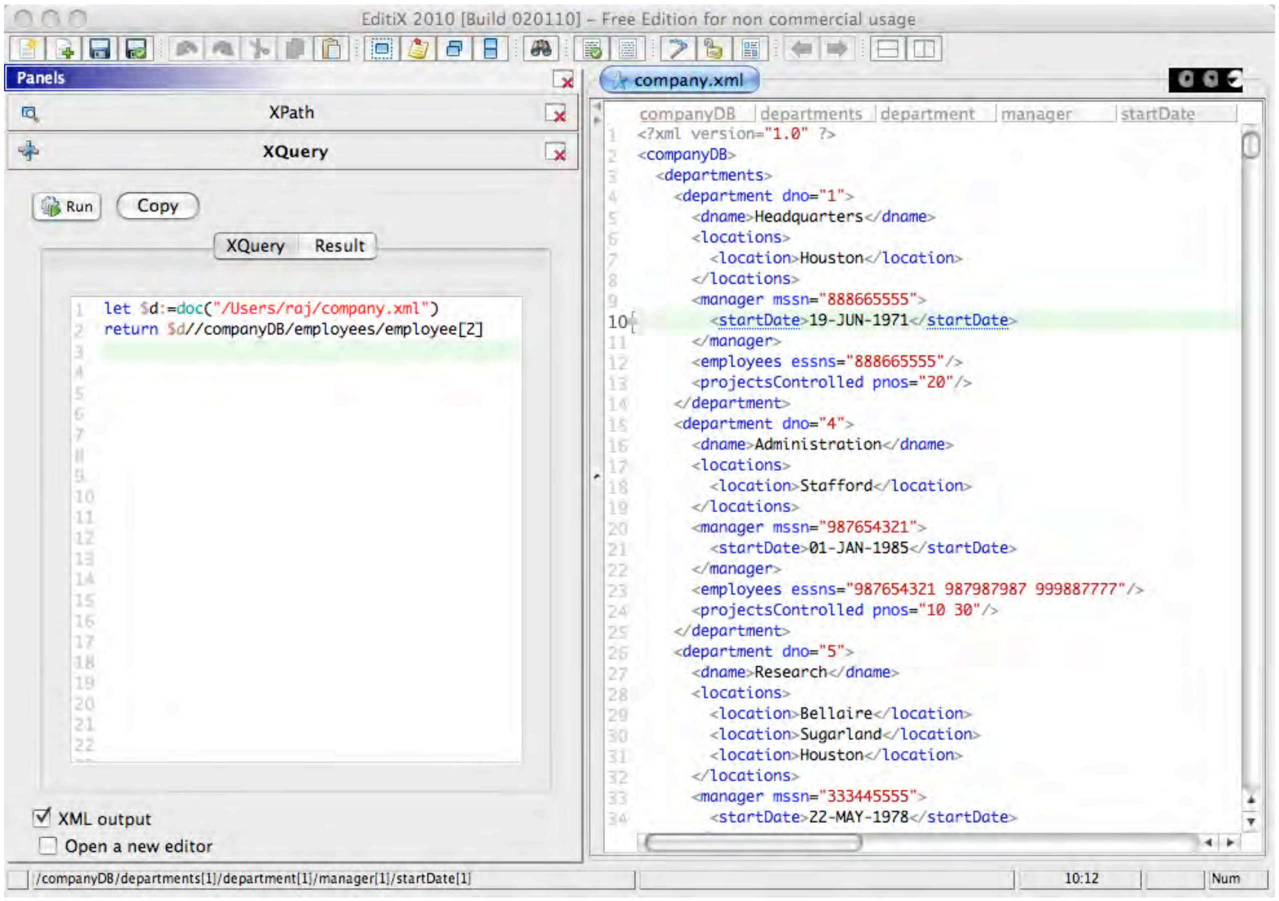
```
/companyDB/employees/employee[@worksFor=ancestor::companyDB/depart
ments/department[dname="Administration"]/@dno]
```

This expression accesses all employee nodes of employees who work for the "`Administration`" department. The use of the `ancestor` axis enables traversing up the XML tree to look for a particular department.

7.5 XQuery

XQuery is the official W3C (World Wide Web Consortium) standard query language for XML data. It is a high-level functional language for formulating ad hoc queries on XML data. The latest XQuery specifications are available at http://www.w3.org/TR/xquery.

Editix provides support for XQuery execution in its user interface. Figure 7.4 shows the editx window with the XQuery tab opened on the left panel.



```
let $d:=doc("/Users/raj/company.xml")
return $d//companyDB/employees/employee[2]
```

```
companyDB | departments | department | manager | startDate
<?xml version="1.0" ?>
<companyDB>
  <departments>
    <department dno="1">
      <dname>Headquarters</dname>
      <locations>
        <location>Houston</location>
      </locations>
      <manager mssn="888665555">
        <startDate>19-JUN-1971</startDate>
      </manager>
      <employees essns="888665555"/>
      <projectsControlled pnos="20"/>
    </department>
    <department dno="4">
      <dname>Administration</dname>
      <locations>
        <location>Stafford</location>
      </locations>
      <manager mssn="987654321">
        <startDate>01-JAN-1985</startDate>
      </manager>
      <employees essns="987654321 987987987 999887777"/>
      <projectsControlled pnos="10 30"/>
    </department>
    <department dno="5">
      <dname>Research</dname>
      <locations>
        <location>Bellaire</location>
        <location>Sugarland</location>
        <location>Houston</location>
      </locations>
      <manager mssn="333445555">
        <startDate>22-MAY-1978</startDate>
```

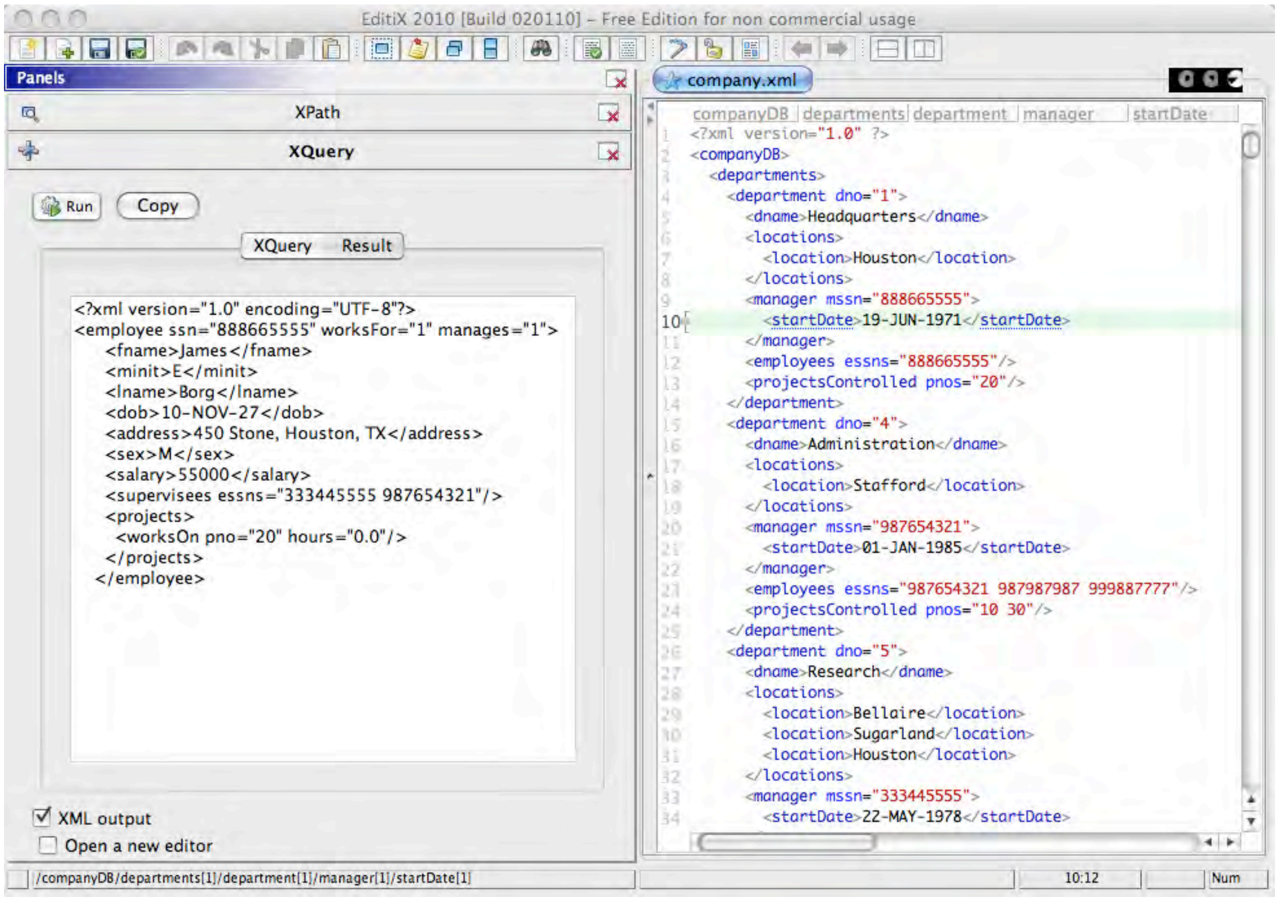
```
/companyDB/departments[1]/department[1]/manager[1]/startDate[1]    10:12    Num
```

Figure 7.4 editix Window with XQuery tab in left panel

The user first enters the query in the text box and then clicks the "Run" button. In the figure the following query is shown:

```
let $d:=doc("/Users/raj/company.xml")
return $d//companyDB/employees/employee[2]
```

The results are displayed under the “Result” tab. To see the results the user needs to click the “Results” tab. A screenshot of the results tab is shown in Figure 7.5.


EditiX 2010 [Build 020110] – Free Edition for non commercial usage
Panels
XPath
XQuery
Run
Copy
XQuery
Result
[✓] XML output
[ ] Open a new editor
company.xml
companyDB departments department manager startDate
10:12
Num

```
<?xml version="1.0" encoding="UTF-8"?>
<employee ssn="888665555" worksFor="1" manages="1">
    <fname>James</fname>
    <minit>E</minit>
    <lname>Borg</lname>
    <dob>10-NOV-27</dob>
    <address>450 Stone, Houston, TX</address>
    <sex>M</sex>
    <salary>55000</salary>
    <supervisees essns="333445555 987654321"/>
    <projects>
      <worksOn pno="20" hours="0.0"/>
    </projects>
  </employee>
```

```
/companyDB/departments[1]/department[1]/manager[1]/startDate[1]
```

```
<?xml version="1.0" ?>
<companyDB>
  <departments>
    <department dno="1">
      <dname>Headquarters</dname>
      <locations>
        <location>Houston</location>
      </locations>
      <manager mssn="888665555">
        <startDate>19-JUN-1971</startDate>
      </manager>
      <employees essns="888665555"/>
      <projectsControlled pnos="20"/>
    </department>
    <department dno="4">
      <dname>Administration</dname>
      <locations>
        <location>Stafford</location>
      </locations>
      <manager mssn="987654321">
        <startDate>01-JAN-1985</startDate>
      </manager>
      <employees essns="987654321 987987987 999887777"/>
      <projectsControlled pnos="10 30"/>
    </department>
    <department dno="5">
      <dname>Research</dname>
      <locations>
        <location>Bellaire</location>
        <location>Sugarland</location>
        <location>Houston</location>
      </locations>
      <manager mssn="333445555">
        <startDate>22-MAY-1978</startDate>
```

Figure 7.5 editix Window with query results in left panel

Note that the XML file is displayed on the right panel, but unlike XPath, the results are not highlighted there. This is because XQuery is a general query language and the results are constructed from the XML document but may not have the same structure as the XML document.

Simple XQuery Expressions

XQuery is a high-level functional language that evaluates expressions of different kinds. Simple arithmetic expressions such as:

```
(2*3)-(8*7)
```

are evaluated by XQuery. Built-in functions can be invoked as shown in the following three examples:

```
concat("Hello"," World")
matches("Monday","^.*da.*$")
current-time()
```

The `concat()` function takes in one or more string arguments and returns the concatenation of all strings. The `matches()` function takes a string input and a second string input representing a regular expression. It returns `true` if the first string matches the regular expression. The `current-time()` function returns the current time of day. All XPath/XQuery built-in functions are documented at http://www.w3.org/TR/xquery-operators/.

XQuery provides the `let` clause to assign values to variables and the `return` clause to construct the output and return the value of the output. Lists in XQuery are flat objects and are constructed by surrounding comma-separated values by parentheses. The following example illustrates list values, the `let` and the `return` clauses, and list aggregate operations:

```
let $list:=(1,5,10,12,15)
return count($list)
```

The above example returns the number of elements in the list. Other list aggregate operations include `avg`, `sum`, `max`, and `min`.

Value comparison operators for primitive data types are: `eq, ne, lt, gt, ge, and le`; general objects (including the primitive types) such as lists etc. can be compared using `<, <=, =, !=, >, >=`. XML nodes comparisons are done with one of the three operators: `<<, >>,` and `is`. The "is" operator compares for exact identity. The document order of nodes is verified by << (appears before) and >> (appears after).

XPath expressions can be directly introduced in the return clause of queries as follows:

```
let $d:=doc("/Users/raj/company.xml")
return $d//companyDB/employees/employee[2]
```

In fact, XPath expressions form the building clocks for constructing XQuery queries and can be used in several clauses as will be discussed later in this section.

Raw XML content also are treated as expressions and simply printed to the output by XQuery.
For example, the following expression is simply sent to output when evaluated by XQuery:

```
<item ino=”222”><iname>Nut</iname><price>22.50</price></item>
```

Curly braces can enclose sub-expressions in XQuery that need to be evaluated before sending to output. The following example employs the curly braces around an expression to print the salaries of employees who work for department number `6`:

```
let $d:=doc("/Users/raj/company.xml")//employee[@worksFor=6]
return
  <dept6Salary>{$d/salary}
  </dept6Salary>
```

FLWOR Expressions

The most important and powerful construct in XQuery is the FLWOR expression. FLWOR, pronounced "flower" stands for `for`, `let`, `where`, `order by`, and `return`, the individual clauses allowed in a query. A FLWOR expression starts with one or more `for` or `let` clauses, followed by an optional `where` clause, followed by an optional `order by` clause, and ending with a `return` clause. FLWOR expressions are illustrated via a series of queries on the `company.xml` document.

Query 1: Get all projects.

```
let $d:=doc("/Users/raj/company.xml")
for $p in $d/companyDB/projects/project
return $p
```

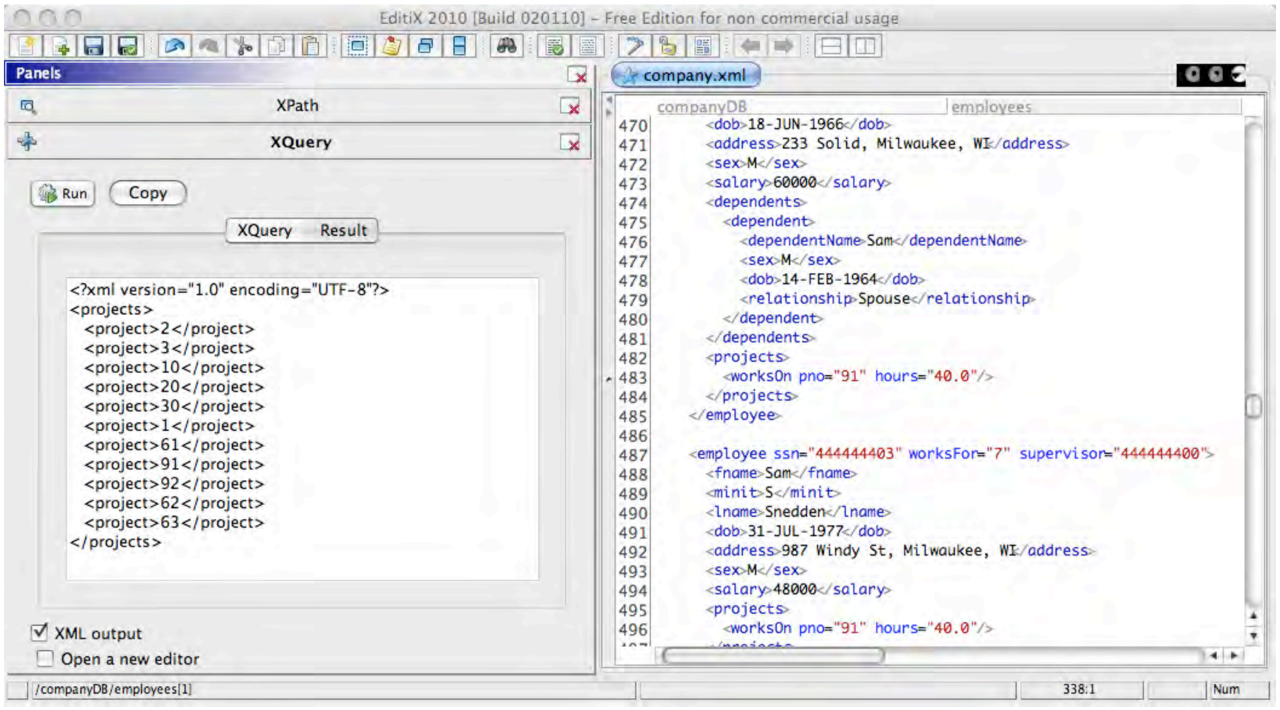
This query uses the `let` expressions to assign the root element of the `company.xml` document to the variable `$d`. Then, in the `for`-clause, the query iterates through all the nodes corresponding to the XPath expression `$d/companyDB/projects/project`. In the `return`-clause the value of the iterator variable `$p` is returned as the result of the query. The result will be a forest of all `<project>` elements.

The next query uses the distinct-values function to remove duplicates:

Query 2: Get distinct project numbers of projects in which employees work.

```
<projects>
{
let $d:=doc("/Users/raj/company.xml")
for $p in distinct-values(
        $d/companyDB/employees/employee/projects/worksOn/@pno)
return
<project>{$p}</project>
}
</projects>
```

In this query, the overall expression is a XML construct which includes FLWOR query within curly braces. The `for`-clause uses the `distinct-values()` function to eliminate duplicates in project numbers found within the `projects/worksOn` sub-element of the `employee` element. The query results for this query are shown in Figure 7.6.



```
<?xml version="1.0" encoding="UTF-8"?>
<projects>
  <project>2</project>
  <project>3</project>
  <project>10</project>
  <project>20</project>
  <project>30</project>
  <project>1</project>
  <project>61</project>
  <project>91</project>
  <project>92</project>
  <project>62</project>
  <project>63</project>
</projects>
```

```
      <dob>18-JUN-1966</dob>
      <address>233 Solid, Milwaukee, WI</address>
      <sex>M</sex>
      <salary>60000</salary>
      <dependents>
        <dependent>
          <dependentName>Sam</dependentName>
          <sex>M</sex>
          <dob>14-FEB-1964</dob>
          <relationship>Spouse</relationship>
        </dependent>
      </dependents>
      <projects>
        <worksOn pno="91" hours="40.0"/>
      </projects>
    </employee>

    <employee ssn="444444403" worksFor="7" supervisor="444444400">
      <fname>Sam</fname>
      <minit>S</minit>
      <lname>Snedden</lname>
      <dob>31-JUL-1977</dob>
      <address>987 Windy St, Milwaukee, WI</address>
      <sex>M</sex>
      <salary>48000</salary>
      <projects>
        <worksOn pno="91" hours="40.0"/>
```

Figure 7.6 editix Window with Query 2 results in left panel

Notice that the project numbers are not ordered. The order by clause allows results to be ordered. By adding the order by clause in Query 2 as shown below, the results can be ordered.

```
<projects>
{
let $d:=doc("/Users/raj/company.xml")
for $p in distinct-values(
            $d/companyDB/employees/employee/projects/worksOn/@pno)
order by number($p)
return
<project>{$p}
</project>
}
</projects>
```

Note the use of the `number()` function in the order by clause. This is necessary because by default the order by clause treats the list as strings and as such the number 100 will appear before 99. The `number()` function converts the string to a number, thereby allowing the ordering to be done on a numeric basis.

Query 3: Get social security numbers of employees whose last name starts with "S".

```
let $d:=doc("/Users/raj/company.xml")
for $e in $d/companyDB/employees/employee
```

```
where starts-with($e/lname,"S")
return <sssn>{$e/@ssn}</sssn>
```

This query calls the built-in function `starts-with()` to determine if the last name of the employee begins with "`S`" and employs the `where`-clause to filter the results as shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<sssn ssn="123456789"/>
<sssn ssn="444444403"/>
<sssn ssn="666666607"/>
```

Query 4: Get last names and first names of employees in the "Research" department.

```
let $d:=doc("/Users/raj/company.xml")
let $r:=$d/companyDB/departments/department[dname="Research"]
for $e in $d/companyDB/employees/employee
where $e/@worksFor=$r/@dno
return
<ResearchEmp>{$e/lname}{$e/fname}</ResearchEmp>
```

This query implements a "join" operation. The query assigns the "`Research`" department object to the variable `$r`. The, in the `for`-clause, the variable `$e` iterates over all employees. The `where`-clause selects only those employees who work for the department denoted by `$r`. The equality comparison works fine since there is only one department with name "Research", i.e. the value for `$r` is a singleton set. The `return`-clause constructs the answer to the query as shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<ResearchEmp>
   <lname>Wong</lname>
   <fname>Franklin</fname>
</ResearchEmp>
<ResearchEmp>
   <lname>Smith</lname>
   <fname>John</fname>
</ResearchEmp>
<ResearchEmp>
   <lname>Narayan</lname>
   <fname>Ramesh</fname>
</ResearchEmp>
<ResearchEmp>
   <lname>English</lname>
   <fname>Joyce</fname>
</ResearchEmp>
```

Query 5: Get employees who work more than 40 hours.

```
let $d:=doc("/Users/raj/company.xml")
for $e in $d/companyDB/employees/employee
where sum($e/projects/worksOn/@hours)>40.0
return
<OverWorkedEmp>{$e/lname}
{$e/fname}<TotalHours>{sum($e/projects/worksOn/@hours)}
</TotalHours>
</OverWorkedEmp>
```

This query illustrates the aggregate operation, `sum`. The for-clause introduces an iterator $e over all employees and the where-clause sums the hours worked on various projects by the employee and check to see if it exceeds 40. The return-clause constructs the results as shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<OverWorkedEmp>
   <lname>Zell</lname>
   <fname>Josh</fname>
   <TotalHours>48</TotalHours>
</OverWorkedEmp>
<OverWorkedEmp>
   <lname>Chase</lname>
   <fname>Jeff</fname>
   <TotalHours>46</TotalHours>
</OverWorkedEmp>
<OverWorkedEmp>
   <lname>Ball</lname>
   <fname>Nandita</fname>
   <TotalHours>44</TotalHours>
</OverWorkedEmp>
<OverWorkedEmp>
   <lname>Bacher</lname>
   <fname>Red</fname>
   <TotalHours>50</TotalHours>
</OverWorkedEmp>
```

Query 6: Get department names and the total number of employees working in the department.

```
let $d:=doc("/Users/raj/company.xml")
for $r in $d/companyDB/departments/department
return
<deptNumEmps>{$r/dname}
<numEmps>{count(tokenize($r/employees/@essns,"\s+"))}
</numEmps>
</deptNumEmps>
```

This example is similar to the previous example, but it illustrates how to tokenize a string of social security numbers separated by a space. The `tokenize()` function takes as its first argument the string of social security numbers of employees working for a particular department and a regular expression denoting the separator. In this case, the regular expression is "`s\+`" indicating one or more spaces. Rest of the query is similar to the previous one. The results are shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<deptNumEmps>
   <dname>Headquarters</dname>
   <numEmps>1</numEmps>
</deptNumEmps>
<deptNumEmps>
   <dname>Administration</dname>
   <numEmps>3</numEmps>
</deptNumEmps>
<deptNumEmps>
   <dname>Research</dname>
   <numEmps>4</numEmps>
</deptNumEmps>
<deptNumEmps>
   <dname>Software</dname>
   <numEmps>8</numEmps>
</deptNumEmps>
<deptNumEmps>
   <dname>Hardware</dname>
   <numEmps>10</numEmps>
</deptNumEmps>
<deptNumEmps>
   <dname>Sales</dname>
   <numEmps>14</numEmps>
</deptNumEmps>
```

Query 7: Get last names of employees who work for a project located in "Houston".

```
<empsWorkingOnHoustonProjects>
{
distinct-values(
let $d:=doc("/Users/raj/company.xml")
for $r in $d/companyDB/projects/project[plocation="Houston"]
return
 for $e in $d/companyDB/employees/employee
 where exists(index-of($r/workers/worker/@essn,$e/@ssn))
 return $e/lname
)
}
</empsWorkingOnHoustonProjects>
```

This query illustrates nesting of FLWOR expressions as well as additional built-in functions. The outer FLWOR expression sets up an iterator `$r` over all "`Houston`" projects. The nested FLWOR expression is present in the `return`-clause and iterates over all `employee` nodes. The `where`-clause checks to see if the social security number of the employee matches one of the worker social security numbers of the "`Houston`" project. The `index-of()` function returns a list of indices where the second argument (search item) appears in the first argument (list to be searched). The `exists()` function returns `true` if its input is non-empty. The results of executing the query are shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<empsWorkingOnHoustonProjects>
Wong Narayan Borg Wallace
</empsWorkingOnHoustonProjects
```

Query 8: Get last names of employees with dependents.

```
let $d:=doc("/Users/raj/company.xml")
for $e in $d/companyDB/employees/employee[dependents]
return $e/lname
```

This query illustrates the use of an XPath-expression (`[dependents]`) as a predicate; employee nodes that have sub-element `<dependents>` are selected and the last names of such employees are returned as the answer to the query.

Query 9: Get last names of employees without dependents.

```
let $d:=doc("/Users/raj/company.xml")
let $empsWithDeps := $d/companyDB/employees/employee[dependents]
for $e in $d/companyDB/employees/employee
where empty(index-of($empsWithDeps,$e))
return $e/lname
```

This query defined a variable `$empsWithDeps` that holds all employee nodes with dependents. Then, the `for`-clause iterates over all employees and selects only those employees who do not appear in the list `$empsWithDeps`.

Query 10: get last names of employees from Milwaukee along with their income group: "Low Income Group" (earning < 40000), "Middle Income Group" (earning between 40000 and 60000), and "High Income Group" (earning more than 80000).

```
<IncomeGroup>
{
let $d:=doc("E:/company.xml")
for $e in
  $d/companyDB/employees/employee[contains(address,"Milwaukee")]
return
<emp>{$e/lname}
<income>
{if  ($e/salary >= 80000) then "High Income"
else if ($e/salary >=60000) then "Middle Income"
else "Low Income"
}
</income>
</emp>
}
</IncomeGroup>
```

This query illustrates the use of conditional expressions in FLWOR queries. The `for`-clause sets up an iterator on all "`Milwaukee`" employee nodes. The `return`-clause constructs the output XML using a conditional expression. The output to the query looks like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<IncomeGroup>
  <emp>
    <lname>Freed</lname>
    <income>High Income</income>
  </emp>
…
…
</IncomeGroup>
```

Query 11: Get employee names of employees who work on all projects located in "Houston".

```
let $d:=doc("/Users/raj/company.xml")
let $houstonProjs :=
 $d/companyDB/projects/project[plocation="Houston"]
for $e in $d/companyDB/employees/employee
where every $p in $houstonProjs satisfies
  (some $q in $e/projects/worksOn satisfies
    $p/@pnumber = $q/@pno)
return concat($e/fname,", ",$e/lname)
```

This query illustrates the use of the "`every`" and "`some`" quantifier constructs in XQuery. The query first computes the list of all projects located in "`Houston`" in the variable `$houstonProjs`. The `for`-clause sets up an iterator on `employee` nodes. The `where`-clause employs the quantifier constructs to verify if every "`Houston`" project is present in the `projects/worksOn` sub-element of the `employee` node. The results are as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
Franklin, Wong
```

The general syntax of the quantifier expressions is:

```
some $var1 in $expr1, …, $varN in $exprN
satisfies $boolExpr

every $var1 in $expr1, …, $varN in $exprN
satisfies $boolExpr
```

The “`some`” expression evaluates to true if at least one assignment of values to the variables result in the Boolean expression being evaluated to true.

The “`every`” expression evaluates to true if all assignment of values to the variables result in the Boolean expression being evaluated to true.

7.6 XML Schema

XML Schema is a schema language for XML documents with a rich set of primitive data types as well as an extensive set of type constructs. The syntax of XML Schema is XML itself, i.e. XML Schemas are themselves well-formed and valid XML documents. The structure of XML documents is defined in XML Schema by the constructing a type system of simple and complex types that describe the elements and sub-elements of the document.

The schema language features are introduced in this section by considering the `company.xml` document introduced earlier in this chapter and creating a schema for it.

Primitive Types

XML Schema provides a host of primitive types including `xs:string`, `xs:integer`, `xs:decimal`, `xs:boolean`, and `xs:date`. Simple elements in the XML document can be defined in the schema as having one of these primitive types. For example,

```
<xs:element name=dname" type="xs:string"/>
```

defines the structure for XML element:

```
<dname>Research</dname>
```

Simple Types

Simple types are used for elements that do not have attributes and that do not have sub-elements. They are also used for attributes. Starting with the basic primitive types, XML Schema provides several constructs to impose restrictions on the values that a particular type can allow from the domain of the primitive types.

Consider the XML element `<dno>6</dno>` and the restriction that the department number be in the range 1 through 50. The XML Schema code that describes the type of the element is shown below:

```
<xs:simpleType name="dnoType">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="1"/>
    <xs:maxInclusive value="50"/>
  </xs:restriction>
</xs:simpleType>
```

Here, the simple type `dnoType` is being defined as a restriction on the base type `xs:integer` with the minimum value being `1` and the maximum value being `50`. The <dno> element is then described in the schema as:

```
<xs:element name=dno" type="dnoType"/>
```

The social security number is restricted to be a 9-digit number and the simple type for it is shown below:

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

Here, the base type is `xs:string` and the restriction is based on a regular expression pattern which indicates that there should be exactly `9` digits. The regular expressions that describe the pattern are very similar to that of Unix regular expressions. Detailed descriptions of the various features of XML Schema including that of the regular expressions can be found at the W3C website: http://www.w3.org/XML/Schema.html

The number of hours per week an employee of the company may work for a project is a decimal number with two digits after the decimal point and occupying a total of 5 spaces. The type is defined as follows:

```
<xs:simpleType name="hoursType">
  <xs:restriction base="xs:decimal">
    <xs:totalDigits value="5"/>
    <xs:fractionDigits value="2"/>
  </xs:restriction>
</xs:simpleType>
```

Enumerated types are possible in XML Schema where the values are chosen from a given set. For example, the `genderType` used in the company example for employees as well as dependents can be defined as follows:

```
<xs:simpleType name="genderType">
   <xs:restriction base="xs:string">
```

```
      <xs:enumeration value="M"/>
      <xs:enumeration value="F"/>
   </xs:restriction>
</xs:simpleType>
```

Lists of Simple Types

XML Schema provides rich support for creating lists of simple types and enforcing several constraints in lists. The following is a list type definition for a list of social security numbers:

```
<xs:simpleType name="listOfSSNType">
  <xs:list itemType="ssnType"/>
</xs:simpleType>
```

Restrictions on lists can be expressed using the facets (a fancy name for restrictions!) `xs:length`, `xs:minLength`, and `xs:maxLength`. For example, to define lists of social security numbers of length `8`, the following code can be used:

```
<xs:simpleType name="eightListOfSSNType">
  <xs:restriction base="listOfSSNType">
    <xs:length value=”8”/>
  </xs:restriction>
</xs:simpleType>
```

Complex Types

Elements that have attributes or those that have sub-elements are said to have complex types and XML Schema provides the ability to compose complex types from simple types. For example, the following element found within the `<employee>` element describes a dependent:

```
<dependent>
  <dependentName>Abner</dependentName>
  <sex>M</sex>
  <dob>29-FEB-1932</dob>
  <relationship>Spouse</relationship>
</dependent>
```

The complex type, `dependentType`, shown below describes the structure of the `<dependent>` element:

```
<xs:complexType name="dependentType">
  <xs:sequence>
    <xs:element name="dependentName" type="xs:string"/>
    <xs:element name="sex" type="genderType"/>
    <xs:element name="dob" type="xs:string"/>
    <xs:element name="relationship" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

The complex type describes the structure as containing four sub-elements in a particular order. The `<xs:sequence>` construct specifies that the order of the sub-element is as specified in the type definition. To define the `<dependents>` element that includes one or more `<dependent>` elements, another complex type can be defined as follows:

```
<xs:complexType name="dependentsType">
  <xs:sequence>
    <xs:element name="dependent" type="dependentType"
                minOccurs="1" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

Note the `minOccurs` and `maxOccurs` attributes that specify that there can be one or more occurrences of the `<dependent>` sub-elements within the `<dependents>` element.

A similar type definition is made for the `<locations>` element within the `<department>` element as follows:

```
<xs:complexType name="locationsType">
  <xs:sequence>
    <xs:element name="location" type="xs:string"
                minOccurs="1" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

A sample XML fragment that conforms to the above type definition is:

```
<locations>
  <location>Atlanta</location>
  <location>Sacramento</location>
</locations>
```

Consider the following XML fragment from the company XML file:

```
<manager mssn="111111100">
  <startDate>15-MAY-1999</startDate>
</manager>
```

This describes a manager and includes an attribute as well as a sub-element. A complex type to describe such a structure is shown below:

```
<xs:complexType name="managerType">
  <xs:sequence>
```

```
    <xs:element name="startDate" type="xs:string"/>
  </xs:sequence>
  <xs:attribute name="mssn" type="ssnType"/>
</xs:complexType>
```

Attributes are described after all the sub-elements are described. In this case there is only one sub-element, `startDate`.

Elements that have attributes and have empty content (i.e. no sub-elements) are also classified under complex types. For example, consider the following element in the `<department>` element.

```
<projectsControlled pnos="61 62 63"/>
```

The complex type definition for this element is shown below:

```
<xs:complexType name="projsControlType">
  <xs:attribute name="pnos" type="listOfPnoType"/>
</xs:complexType>
```

Elements with Simple Content and Attributes

Describing the structure of an element with simple content and with attributes is done by using the `<xs:simpleContent>` construct in XML Schema. Such an element appears in the company XML document as follows:

```
<worker essn="555555500">40.0</worker>
```

The complex type to describe this element is shown below:

```
<xs:complexType name="workerType">
  <xs:simpleContent>
    <xs:extension base="hoursType">
      <xs:attribute name="essn" type="ssnType"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

Within the `<simpleContent>` construct, the `hoursType` type is extended by adding an attribute. Now, to enclose one or more `<worker>` elements within the `<workers>` element such as:

```
<workers>
  <worker essn="555555500">40.0</worker>
  <worker essn="555555501">44.0</worker>
  <worker essn="666666605">40.0</worker>
```

```
</workers>
```

the following complex type is defined:

```
<xs:complexType name="workersType">
  <xs:sequence>
    <xs:element name="worker" type="workerType"
                minOccurs="1" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

Complete XML Schema File

The complete XML Schema file for the company document is constructed as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  definitions of all simple and complex types
  …
  …
  <xs:element name="companyDB" type="companyDBType"/>
</xs:schema>
```

The file starts with the XML version statement followed by the `<xs:schema>` element. Within this element, all simple and complex types are defined. Finally the root element `<companyDB>` is defined. The entire schema file (`company.xsd`) along with the XML document file (`company.xml`) is available along with this lab manual.

Exercises

NOTE: All references to Exercises and Laboratory Exercises in the following problems refer to the numbering in the Elmasri/Navathe text.

1. Write and execute XQuery expressions for the following queries on the company.xml document:
    a. Retrieve the names and addresses of employees who work for the "Research" department.
    b. For every project located in "Stafford", retrieve the project number, the controlling department number, and the department's manager's last name, address, and birth date.
    c. Retrieve the names of all employees who have two or more dependents.
    d. Retrieve the names of managers who have at least one dependent.
    e. Retrieve the names of employees who work on all projects controlled by department "5".
2. Write and execute XQuery expressions for all queries of laboratory exercise 6.34 on page 192 of the Elmasri/Navathe textbook (6th edition).

3. Consider the mail order database described in problem 2 of the Exercises in Chapter 1 of this lab manual.
    a. Create a XML representation of the data described there (you should invent your own data instances).
    b. Write a XML Schema specification for the XML document constructed in part a.
    c. Write expressions in XQuery to answer all queries of problem 2 of the Exercises in Chapter 2 of this lab manual.
4. Consider the grade book database described in problem 6 of the Exercises in Chapter 1 of this lab manual.
    a. Create a XML representation of the data described there (you should invent your own data instances).
    b. Write a XML Schema specification for the XML document constructed in part a.
    c. Write expressions in XQuery to answer all queries of problem 3 of the Exercises in Chapter 2 of this lab manual.
5. Consider the bibliography XML document, bib.xml, provided along with this lab manual.
    a. Write a XML Schema specification for the XML document.
    b. Write expressions in XQuery to answer the following queries:
        i. Find articles that have "Temporal" in their titles.
        ii. Find all articles authored by "Raghu Ramakrishnan".
        iii. Find number of articles with more than 3 authors.
        iv. Produce a listing of URLs of articles for each author. Output should consist of author names followed by list of URLS, sorted by author names.
        v. Find all articles that are over 40 pages long.