XQuery

XQuery is a declarative programming language that can be used to extract information from an XML document in much the same way as SQL extracts information from a relational database.

XQuery queries can act on a single XML document or a fixed collection of XML documents.

The results of an XQuery query are normally expressed using XML syntax, although the language of XQuery itself is not expressed as XML.

Comments in XQuery are delimited by "(:" and ":)".

XQuery Data Model

XQuery incorporates a rich set of primitive data types, including the atomic types from XPath and the simple types from the XML Schema language.

It recognizes seven node types: document, element, attribute, text, namespace, processing-instruction, and comment.

Sequences

The main structure of XQuery is the ordered sequence.

The sequence in XQuery corresponds to the node set in XSLT, but they have different characteristics.

A sequence may contain items of two types: simple types or nodes.

A sequence may not contain another sequence.

A sequence with one item is the same as the item itself.

Sequences may contain items of various types (they are not homogeneous).

Sequence literals are delimited by parentheses, and use commas to separate their items.

The *to* operator can be used to define a sequence.

Sequences are not sets; their items have ordinal position, starting at 1, and may include duplicates.

Items may be defined using data type constructors as in xs:integer("837") and xs:date("2001-01-01").
Note that the month and day specifications require two digits.

Examples of Sequences

```
(1, 3, 5, 7, 9, 11,13)

1 to 5                                  (: same as (1, 2, 3, 4, 5) :)

(1, 1, 2, 3, 3, 3, 4, 5, 5, 5, 5)       (: not same as previous
                                                sequence :)

(xs:date("2005-05-02"), xs:date("2005-05-04"),
                                        xs:date("2005-05-06"))

("Hello", 45.8, 994, xs:date("1922-11-11"), true())

()                                      (: Empty sequence :)

(<a>herky</a>, <b/>, <c></c>, <d>dodo</d>)

((1, 3, 5), (7, 9, 11), 13, ())         (: same as first sequence :)

(xs:boolean("true"), xs:float("2.58"), 'Herky', xs:boolean(1))
```

XQuery Expressions

XQuery is a language of expressions, each of which produces an XQuery sequence.

Remember that a single item is identical to a singleton sequence containing that item.

Variables in XQuery are XML qualified names preceded by $.

Varieties of XQuery Expressions

- Literals and Variables
- Expressions with operators
- Calls of predefined and user defined functions
- Conditional expressions (*if-then-else*)
- Path expressions
- Element and Attribute constructors
- Quantified expressions (*every* and *some*)
- FLWOR expressions

Examples: Simple Expressions

```
483

34 + 17 * 5

$m div 2                (: $m must be bound to a number :)

string-length("herky")

if ($m > 0) then $m - 1 else $m + 1
                        (: must be a space between m and - :)
```

Path Expressions

XQuery allows XPath location paths as expressions.

We can build a literal node and select a subnode.

```
<one><two>herky</two><two>hawk</two></one>/two/text()
```

produces the value *herkyhawk*.

We want to use these paths to explore an existing XML document.

One way to specify an XML document in XQuery is the function *doc*, which takes the name of a file as a string parameter.

This function returns a single document as a source tree.

Example: Fetch *student* elements

```
doc("roster.xml")/roster/students/student
```

Running XQuery

The best (free) implementation of XQuery that I found was from Saxon (Michael Kay).

Install the jar file *saxon8.8.jar* on your machine by putting it in your *bin* directory and pointing CLASSPATH to its location.

The XQuery expression needs to be entered in a file to be processed by Saxon.

For example, place the previous example into a text file with the name *students*.

Execution

```
% java net.sf.saxon.Query students
<?xml version="1.0" encoding="UTF-8"?>
  <student id="94811">
    <name>Rusty Nail</name>
    <quizzes>
          <quiz>16</quiz>
          <quiz>12</quiz>
    </quizzes>
    <projects>
          <project>44</project>
          <project>52</project>
    </projects>
    <exams>
          <exam>77</exam>
          <exam>68</exam>
          <exam>49</exam>
    </exams>
  </student>

  <student id="2562">
    <name>Guy Wire</name>
    <quizzes>
          <quiz>15</quiz>
          <quiz>23</quiz>
    </quizzes>
    <projects>
          <project>33</project>
          <project>47</project>
    </projects>
    <exams>
          <exam>78</exam>
          <exam>86</exam>
          <exam>88</exam>
    </exams>
  </student>

      :
```

```
<student id="137745">
  <name>Barb Wire</name>
  <quizzes>
        <quiz>20</quiz>
        <quiz>25</quiz>
  </quizzes>
  <projects>
        <project>48</project>
        <project>60</project>
  </projects>
  <exams>
        <exam>38</exam>
        <exam>48</exam>
        <exam>66</exam>
  </exams>
</student>%
```

Note that complete elements are returned as the value of the expression, and that the result is not a well-formed XML document because it does not have a single root.

Such a structure is called a *forest*.

You can see that Saxon always provides an XML declaration at the beginning of the result.

Furthermore, it does not supply a new line at the end of the result.

To stay in line with Saxon, we will define our queries to produce legal XML as much as possible.

That means intermixing XML tags and XPath expressions that need to be evaluated.

To separate literal data from expressions that must be evaluated, XQuery uses braces to indicate what must be evaluated.

I use an alias *xquery* that takes the file name as a parameter and causes the execution of the Java class net.sf.saxon.Query.

Example: Fetch name elements

```
<newRoot>
{
    doc("roster.xml")/roster/students/student/name
}
</newRoot>
```

Results

```
<?xml version="1.0" encoding="UTF-8"?>
<newRoot>
  <name>Rusty Nail</name>
  <name>Guy Wire</name>
  <name>Norman Conquest</name>
  <name>Eileen Dover</name>
  <name>Barb Wire</name>
</newRoot>%
```

Example: Fetch the names only

```
<newRoot>
{
    doc("roster.xml")/roster/students/student/name/text()
}
</newRoot>
```

Results

```
<?xml version="1.0" encoding="UTF-8"?>
<newRoot>Rusty NailGuy WireNorman ConquestEileen
                                DoverBarb Wire</newRoot>%
```

Alternate Version

```
<newRoot>
{
   doc("roster.xml")/roster/students/student/string(name)
}
</newRoot>
```

The results are the same.

Saxon Options

The results from an XQuery query under Saxon are displayed on the screen.

How to Put Results in a File

Saxon has an option (-o outfile) that directs the results to the named file.

```
% java net.sf.saxon.Query -o stdts.out students
```

Another way is to pipe the stream through the Unix *cat* function.

```
% java net.sf.saxon.Query students l cat > stds.out
```

To see the other Saxon options enter:

```
% java net.sf.saxon.Query
```

FLWOR

XQuery queries can be written as FLWOR (pronounced "flower") expressions.

F : *for* each item *in* an XPath expression

L : *let* a new variable have a value

W : *where* a condition (a predicate) is satisfied

O : *order* by some XPath value

R : *return* a sequence of values

Example 1

```
for $m in 1 to 10
let $n := $m + 1
where $m > 4
order by $m descending
return $m * $n
```

Produces the result: 110 90 72 56 42 30

Observe that the *let* phrase uses := to describe the binding of a variable. This symbol is the assignment operator in many programming languages.

As in XSLT, variables may not have their values altered.

Some of the components of a FLWOR expression are optional.

A *return* is required, and it must be preceded by at least one *for* or one *let*.

A FLOWR expression may have multiple *for* and *let* components, and these can be nested in the expression.

We want to produce well-formed XML, so delimit the sequence and its components with XML tags.

Remember braces indicate which parts of the expression need to be evaluated.

Example 2

```
<numbers>
{
    for $m in (1 to 10)       (: parentheses are redundant :)
    let $n := $m + 1
    where $m > 4
    order by $m descending
    return  <num>{ $m * $n }</num>
}
</numbers>
```

Results

```
<?xml version="1.0" encoding="UTF-8"?>
<numbers>
  <num>110</num>
  <num>90</num>
  <num>72</num>
  <num>56</num>
  <num>42</num>
  <num>30</num>
</numbers>%
```

We now turn to using XQuery to solve the problems that were considered in the XSLT chapter.

You can compare these solutions with the XSLT stylesheets to help you judge the usefulness of XQuery.

Problem 1

Find the sum of all first exams, the number of first exams, and the average on that exam.

File: p1.xq

```
<exam1>
{
    let $doc := doc("roster.xml")
    let $e1s := $doc/roster/students/student/exams/exam[1]
    let $sum := sum($e1s)
    let $num := count($e1s)
    return (<sum>{ $sum }</sum>,
            <num>{ $num }</num>,
            <average>{ $sum div $num }</average>)
}
</exam1>
```

Observe how a sequence of elements is built using parentheses and commas in the return expression.

Also, note how braces are used to force evaluation when expressions are interspersed with literal output.

Results from p1.xq

```
<?xml version="1.0" encoding="UTF-8"?>
<exam1>
  <sum>382</sum>
  <num>5</num>
   <average>76.4</average>
</exam1>
```

Problem 2

Find the sum, the count, and the average for the quizzes for each student.

An XQuery program is generally made up of a *prolog* and a *body*.

So far our examples have only consisted of the body.

A prolog can be used to define global variables, functions, and namespaces among a few other items.

Each prolog definition begins with the keyword *declare* and ends with a semicolon.

In the next solution we define a global variable in the prolog.

File: p2.xq

```
declare variable $doc := doc("roster.xml");
<quizzes>
{
    for $s in $doc/roster/students/student
    let $sum := sum($s/quizzes/quiz)
    let $num := count($s/quizzes/quiz)
   return
   <student>
        { $s/name }
        <sum>{ $sum }</sum>
        <num>{ $num }</num>
        <average>{ $sum div $num }</average>
   </student>
}
</quizzes>
```

In this solution, the XPath "$s/name" returns an element, so we do not need to supply tags, but tags are required for the three new elements, *sum*, *num*, and *average*.

Results from p2.xq

```
<?xml version="1.0" encoding="UTF-8"?>
<quizzes>
  <student>
    <name>Rusty Nail</name>
    <sum>28</sum>
    <num>2</num>
     <average>14</average>
  </student>
  <student>
    <name>Guy Wire</name>
    <sum>38</sum>
    <num>2</num>
     <average>19</average>
  </student>
  <student>
     <name>Norman Conquest</name>
    <sum>44</sum>
    <num>2</num>
     <average>22</average>
  </student>
  <student>
    <name>Eileen Dover</name>
    <sum>39</sum>
    <num>2</num>
     <average>19.5</average>
  </student>
  <student>
    <name>Barb Wire</name>
    <sum>45</sum>
    <num>2</num>
     <average>22.5</average>
  </student>
</quizzes>
```

Problem 3

Find out if there are exam scores below 50. If so, tell how many there are.

This problem makes good use of the conditional expression.

Do not confuse a conditional expression (if-then-else) with the conditional commands (**if** and **if-else**) in Java.

File: p3.xq

```
declare variable $doc := doc("roster.xml");
<exams>
{
    let $num := count($doc/roster/students/student/
                                              exams/exam[. < 50])
    return
    <answer>
    {
        if ($num > 0)
            then concat($num, " exams below 50")
            else "No exams less than 50."
    }
    </answer>
}
</exams>
```

Results from p3.xq

```
<?xml version="1.0" encoding="UTF-8"?>
<exams>
  <answer>3 exams below 50</answer>
</exams>
```

Comparison Operators (four kinds)

So far our use of comparison operators has been naive, simply comparing values with the traditional symbols <, >, and their variants.

Xquery has a much more sophisticated range of comparison operators that we cover now.

Value Comparison Operators

These operators are used to compare single values and sequences of single or no values.

They produce a boolean value (true or false), the empty sequence, or an error.

The types of the operands must be compatible, although some types will be promoted to yield compatible types.

Unfortunately, string does not promote to a number type automatically.

The value comparison operators:

eq ne lt le gt ge

Note:

Using comparison operators can be very tricky.

Comparing the contents of elements will normally be made as string comparisons.

If we want to compare values as numbers, we must convert the strings to numbers.

Problem 4

Find the names of all students who scored higher on exam 1 than on exam 3.

This problem can be solved by using a conditional expression or by using a where clause in the FLWOR expression.

File: p4a.xq

```
<higher>
{
    for $s in doc("roster.xml")/roster/students/student
    let $e1 := $s/exams/exam[1]
    let $e3 := $s/exams/exam[3]
    return
       if ($e1 gt $e3)              (: beware :)
          then
            <student>
            { ($s/name, $s/exams/exam[1], $s/exams/exam[2]) }
            </student>
          else ()
}
</higher>
```

When the test fails, the query returns an empty sequence, which is equivalent to returning nothing.

File: p4b.xq

```
<higher>
{
    for $s in doc("roster.xml")/roster/students/student
    let $e1 := $s/exams/exam[1]
    let $e3 := $s/exams/exam[3]
    where $e1 gt $e3                    (: beware :)
    return
       <student>
       { ($s/name, $s/exams/exam[1], $s/exams/exam[2]) }
       </student>
}
</higher>
```

Change Rusty Nail's first exam to 7.

Results from p4a.xq and p4b.xq

```
<?xml version="1.0" encoding="UTF-8"?>
<higher>
  <student>
    <name>Rusty Nail</name>
    <exam>7</exam>
    <exam>49</exam>
  </student>
  <student>
    <name>Norman Conquest</name>
    <exam>99</exam>
    <exam>78</exam>
  </student>
  <student>
    <name>Eileen Dover</name>
    <exam>90</exam>
    <exam>89</exam>
  </student>
</higher>
```

The problem with these queries is that the two exams are being compared as strings. The value comparison operator *gt* does not promote the values to integers.

Solution: Cast each value to an integer explicitly.

if (xs:integer($e1) gt xs:integer($e3)) in *p4a.xq*

where xs:integer($e1) gt xs:integer($e3) in *p4b.xq*

Now the queries work correctly.

Problem 5

Find all students who scored 80 or higher on exam 3.

File: p5.xq

```
<exams>
{
    for $s in doc("roster.xml")/roster/students/student
    let $e := $s/exams/exam[3]
    where $e ge 80                (: beware :)
    return
        <student>
            { $s/name }
            <exam>{ $e/text() }</exam>
        </student>
}
</exams>
```

Alternative

```
        <student>{ ($s/name, $e) }</student>
```

Results from p5vc.xq

```
Warning: on line 12 of file:
   /mnt/nfs/fileserv/fs3/slonnegr/xml/xquery/roster/p5vc:
      Comparison of xdt:untypedAtomic? to xs:integer will
      fail unless the first operand is empty

Error on line 12 of
file:/mnt/nfs/fileserv/fs3/slonnegr/xml/xquery/roster/p5vc:
  XPTY0004: Cannot compare xs:string to xs:integer
  Query processing failed: Run-time errors were reported
```

The problem with this query is that the exam as a string is being compared to a number. The value comparison operator *ge* does not promote this string value to an integer.

Solution: Cast the string value to an integer explicitly.

```
        where xs:integer($e) ge 80
```

Now the query works correctly.

Results from p5.xq (revised)

```
<?xml version="1.0" encoding="UTF-8"?>
<exams>
  <student>
    <name>Guy Wire</name>
    <exam>88</exam>
  </student>
  <student>
    <name>Eileen Dover</name>
    <exam>89</exam>
  </student>
</exams>
```

General Comparisons

These operators are used to compare two sequences.

They return true if any pair of elements from the two sequences satisfy the relation.

If the sequences are singleton values, these comparisons will be similar to value comparison operators.

The general comparison operators:

= != < <= > >=

Examples

Expression	Value
(1, 2, 3) = (3, 4)	true
(1, 2, 3) != (3, 4)	true
(1, 2, 3) >= (3, 4)	true
(1, 2, 3) < (3, 4)	true
(1, 2) = (3, 4)	false

These general comparison operators can be used in many examples that compare two single values, but remember that comparisons of strings will be carried out alphabetically.

Example: p4a.xq

if (xs:integer($e1) gt xs:integer($e3)) can be written

```
if (xs:integer($e1) > xs:integer($e3))
```

Example: p4b.xq

where xs:integer($e1) gt xs:integer($e3) can be written

```
where xs:integer($e1) > xs:integer($e3)
```

Example: p5.xq

where xs:integer($e) ge 80 can be written

```
where xs:integer($e) >= 80
```

Node Comparison: *is*

The *is* operator is used to compare single nodes and empty sequences.

This operator tests for node identity in the same way that the Java == operator tests for identity between objects.

Examples

Expression	Value
<tag>content</tag> is <tag>content</tag>	false
let $x := <tag>content</tag> let $y := $x return $x is $y	true
doc('roster.xml') is doc('roster.xml'))	true
(1, 2) is (1, 2)	*error*

Problem

Produce an XML document that contains each pair of distinct students in *roster.xml*.

Since this property describe a symmetric relation, we get each pair twice in different orders.

File: pairs.xq

```
declare variable $doc := doc("roster.xml");
<answer>
{
    for $s1 in $doc/roster/students/student
    for $s2 in $doc/roster/students/student
    where not($s1 is $s2)
    return
        <pair>
        {  ($s1/name, $s2/name)  }
        </pair>
}
</answer>
```

Results from pairs.xq (20 pairs)

```
<?xml version="1.0" encoding="UTF-8"?>
<answer>
  <pair>
    <name>Rusty Nail</name>
    <name>Guy Wire</name>
  </pair>
  <pair>
    <name>Rusty Nail</name>
    <name>Norman Conquest</name>
  </pair>
  <pair>
    <name>Rusty Nail</name>
    <name>Eileen Dover</name>
  </pair>
  <pair>
    <name>Rusty Nail</name>
    <name>Barb Wire</name>
  </pair>
  <pair>
    <name>Guy Wire</name>
    <name>Rusty Nail</name>
  </pair>
              :
              :
              :
  <pair>
    <name>Eileen Dover</name>
    <name>Barb Wire</name>
  </pair>
  <pair>
    <name>Barb Wire</name>
    <name>Rusty Nail</name>
  </pair>
  <pair>
    <name>Barb Wire</name>
    <name>Guy Wire</name>
  </pair>
```

```
    <pair>
      <name>Barb Wire</name>
      <name>Norman Conquest</name>
    </pair>
    <pair>
      <name>Barb Wire</name>
      <name>Eileen Dover</name>
    </pair>
  </answer>
```

Node Comparison: *deep-equal*

The *deep-equal* function is used to compare single nodes and sequences.

This function traverses the tree rooted at the nodes or the sequences to see if they are identical in structure and values.

Examples

Expression	Value
deep-equal(<tag>123</tag>, <tag>123</tag>)	true
let $v := <tag>123</tag> return deep-equal($v, $v)	true
deep-equal(doc('roster.xml'), doc('roster.xml'))	true
deep-equal((1, 2), (2, 1))	false
deep-equal((1, 2), (1, 2)),	true
deep-equal(<tg a="1">z</tg>, <tg a="2">z</tg>))	false
deep-equal(<tg a="1">z</tg>, <tg a="1">z</tg>))	true

Order Comparison Operators

These operators are used to compare the positions of two nodes in an XML document.

- << returns true if the first operand occurs before the second in the document (the first operand is reachable from the second operand using the *preceding* axis).
- >> returns true if the first operand occurs after the second in the document (the first operand is reachable from the second operand using the *following* axis).

Problem

Produce an XML document that contains each pair of distinct students in *roster.xml*, but produce each pair only once independent of order.

This solution creates the pairs as an asymmetic relation.

File: pairsA.xq

```
declare variable $doc := doc("roster.xml");
<answer>
{
    for $s1 in $doc/roster/students/student
    for $s2 in $doc/roster/students/student
    where $s1 << $s2
    return
        <pair>
        {  ($s1/name, $s2/name)  }
        </pair>
}
</answer>
```

Results from pairsA.xq (10 pairs)

```
<?xml version="1.0" encoding="UTF-8"?>
<answer>
  <pair>
    <name>Rusty Nail</name>
    <name>Guy Wire</name>
  </pair>
  <pair>
    <name>Rusty Nail</name>
    <name>Norman Conquest</name>
  </pair>
  <pair>
    <name>Rusty Nail</name>
    <name>Eileen Dover</name>
  </pair>
  <pair>
    <name>Rusty Nail</name>
    <name>Barb Wire</name>
  </pair>
  <pair>
    <name>Guy Wire</name>
    <name>Norman Conquest</name>
  </pair>
  <pair>
    <name>Guy Wire</name>
    <name>Eileen Dover</name>
  </pair>
  <pair>
    <name>Guy Wire</name>
    <name>Barb Wire</name>
  </pair>
  <pair>
    <name>Norman Conquest</name>
    <name>Eileen Dover</name>
  </pair>
  <pair>
    <name>Norman Conquest</name>
    <name>Barb Wire</name>
  </pair>
```

```
  <pair>
    <name>Eileen Dover</name>
    <name>Barb Wire</name>
  </pair>
</answer>
```

Finding Positions in Sequences

The *for* phrase in a FLWOR expression allows an *at* clause that captures the ordinal position of each item processed by the *for*.

```
for $m at $p in (5, 10, 15, 20)
return ($p, $m)
```

This XQuery program returns the sequence 1 5 2 10 3 15 4 20.

Problem 6

Find all students who scored 80 or higher on any exam, indicating which exam it is.

The following solution uses a variable to remember the position of each exam in the sequence containing the three exams for each student.

File: p6.xq

```
<exams>
{
    for $s in doc("roster.xml")/roster/students/student
    for $e at $pos in $s/exams/exam
    where xs:integer($e) ge 80
    return
        <student>
          { $e/ancestor::student/name  (: or $s/name :)  }
          { $e }
          <position>{ $pos }</position>
        </student>
}
</exams>
```

Results from p6.xq

```
<?xml version="1.0" encoding="UTF-8"?>
<exams>
  <student>
    <name>Guy Wire</name>
    <exam>86</exam>
    <position>2</position>
  </student>
  <student>
    <name>Guy Wire</name>
    <exam>88</exam>
    <position>3</position>
  </student>
  <student>
    <name>Norman Conquest</name>
    <exam>99</exam>
    <position>1</position>
  </student>
  <student>
    <name>Eileen Dover</name>
    <exam>90</exam>
    <position>1</position>
  </student>
  <student>
    <name>Eileen Dover</name>
    <exam>89</exam>
    <position>3</position>
  </student>
</exams>
```

Sorting

The *order by* clause in a FLWOR expression controls the order that the sequence will be output.

It takes an expression that specifies the properties of the sequence items that are used to sort the sequence.

The property specification may be followed by *ascending* (the default) or *descending* modifiers.

Multiple properties can be listed, separated by commas, with the early properties taking precedence over the later properties.

Problem 7

Show the total on projects for each student, listing the students by name, sorted alphabetically.

File: p7a.xq

```
<projects>
{
    for $s in doc("roster.xml")/roster/students/student
    let $proj := sum($s/projects/project)
    order by $s/name
    return
        <student>
            { $s/name }
            <total>{ $proj }</total>
        </student>
}
</projects>
```

Results from p7a.xq

```
<?xml version="1.0" encoding="UTF-8"?>
<projects>
  <student>
    <name>Barb Wire</name>
    <total>108</total>
  </student>
  <student>
    <name>Eileen Dover</name>
    <total>85</total>
  </student>
  <student>
    <name>Guy Wire</name>
    <total>80</total>
  </student>
  <student>
    <name>Norman Conquest</name>
    <total>96</total>
  </student>
  <student>
    <name>Rusty Nail</name>
    <total>96</total>
  </student>
</projects>
```

In a second solution, we sort the names by surname first and then by given name.

File: p7b.xq

```
<projects>
{
    for $s in doc("roster.xml")/roster/students/student
    let $proj := sum($s/projects/project)
    order by substring-after($s/name,' '),
                substring-before($s/name,' ')
```

```
    return
        <student>
            { $s/name }
            <total>{ $proj }</total>
        </student>
}
</projects>
```

Results from p7b.xq

```
<?xml version="1.0" encoding="UTF-8"?>
<projects>
  <student>
    <name>Norman Conquest</name>
    <total>96</total>
  </student>
  <student>
    <name>Eileen Dover</name>
    <total>85</total>
  </student>
  <student>
    <name>Rusty Nail</name>
    <total>96</total>
  </student>
  <student>
    <name>Barb Wire</name>
    <total>108</total>
  </student>
  <student>
    <name>Guy Wire</name>
    <total>80</total>
  </student>
</projects>
```

In the next version, we sort the students by their ID numbers.

File: p7c.xq

```
<projects>
{
    for $s in doc("roster.xml")/roster/students/student
    let $proj := sum($s/projects/project)
    order by $s/@id
    return
        <student>
            <id>{ data($s/@id) }</id>
            <total>{ $proj }</total>
        </student>
}
</projects>
```

Results from p7c.xq

```
<?xml version="1.0" encoding="UTF-8"?>
<projects>
  <student>
    <id>132987</id>
    <total>96</total>
  </student>
  <student>
    <id>137745</id>
    <total>108</total>
  </student>
  <student>
    <id>2562</id>
    <total>80</total>
  </student>
  <student>
    <id>49194</id>
    <total>85</total>
  </student>
  <student>
    <id>94811</id>
    <total>96</total>
  </student>
</projects>
```

Observe that we used the function *data* to extract the value of the *id* attribute. The function *string* has the same effect.

Without this function application, the attribute *id* stays an attribute, and the first *id* element will be presented as shown below.

```
<id id="132987"/>
```

The only problem with the solution given above is that the ID numbers are sorted alphabetically instead of numerically.

To get a numeric sort we need to convert the *order by* property specified by the *id* attribute into an integer.

File: p7d.xq

```
<projects>
{
    for $s in doc("roster.xml")/roster/students/student
    let $proj := sum($s/projects/project)
    order by xs:integer($s/@id)
    return
        <student>
            <id>{ string($s/@id) }</id>
            <total>{ $proj }</total>
        </student>
}
</projects>
```

Results from p7d.xq

```
<?xml version="1.0" encoding="UTF-8"?>
<projects>
  <student>
    <id>2562</id>
    <total>80</total>
  </student>
```

```
  <student>
    <id>49194</id>
    <total>85</total>
  </student>
  <student>
    <id>94811</id>
    <total>96</total>
  </student>
  <student>
    <id>132987</id>
    <total>96</total>
  </student>
  <student>
    <id>137745</id>
    <total>108</total>
  </student>
</projects>
```

Problem 8

Find the average scores on the exams for each of the students.

For a change of pace, we sort the student names by surname from the end of the alphabet.

File: p8.xq

```
<exams>
{
    for $s in doc("roster.xml")/roster/students/student
    let $avg := avg($s/exams/exam)
    order by substring-after($s/name,' ') descending
    return
        <student>
            { $s/name }
            <average>{ $avg }</average>
        </student>
}
</exams>
```

Results from p8.xq

```
<?xml version="1.0" encoding="UTF-8"?>
<exams>
  <student>
    <name>Guy Wire</name>
    <average>84</average>
  </student>
  <student>
    <name>Barb Wire</name>
    <average>50.66666666666664</average>
  </student>
  <student>
    <name>Rusty Nail</name>
    <average>64.66666666666667</average>
  </student>
  <student>
    <name>Eileen Dover</name>
    <average>86</average>
  </student>
  <student>
    <name>Norman Conquest</name>
    <average>85.33333333333333</average>
  </student>
</exams>
```

Function Definitions

The prolog of an XQuery program allows the definition of functions as well as variables.

Basic Syntax

```
declare function funName($p1, $p2, $p3)
{
      an expression that defines the result of the
      function
};                (: do not forget the semicolon :)
```

The Saxon implementation of XQuery requires that all user-defined functions must lie in an explicit namespace.

A namespace and its prefix are defined in the prolog using the following syntax.

```
declare namespace prefix = "unique.uri";
```

In the next solution to Problem 8, we define a namespace and a function in that namespace to calculate the average of a sequence of nodes.

File: p8f.xq

Suppose for the sake of this example that XQuery has no function to compute the average of a sequence of numbers.

In this solution we will write our own function for this computation.

```
declare namespace myfun = "myfun.slonnegr.cs.uiowa.edu";
declare function myfun:average($nodes)
{
    let $sum := sum($nodes)
    let $num := count($nodes)
    return $sum div $num
};

<exams>
{
    for $s in doc("roster.xml")/roster/students/student
    let $avg := myfun:average($s/exams/exam)
    order by substring-after($s/name, ' ') descending
    return
        <student>
            { $s/name }
            <average>{ $avg }</average>
        </student>
}
</exams>
```

The results are the same as those with the previous solution to Problem 8.

Creating Elements and Attributes

We have seen many examples of XQuery programs that have created elements for the result document using literal text to define opening and closing tags.

Attributes can be associated with these elements defined literally by putting them into the start tags.

As an example, here is the body of the previous XQuery program with an attribute defined for the student elements.

File: p8fa.xq

```
<exams>
{
    for $s at $p in doc("roster.xml")/roster/students/student
    let $avg := myfun:average($s/exams/exam)
    order by substring-after($s/name, ' ') descending
    return
        <student index="{ $p }">
            { $s/name }
            <average>{ $avg }</average>
        </student>
}
</exams>
```

Results from p8fa.xq

```
<?xml version="1.0" encoding="UTF-8"?>
<exams>
  <student index="2">
    <name>Guy Wire</name>
    <average>84</average>
  </student>
  <student index="5">
    <name>Barb Wire</name>
    <average>50.66666666666664</average>
  </student>
  <student index="1">
    <name>Rusty Nail</name>
    <average>64.66666666666667</average>
  </student>
  <student index="4">
    <name>Eileen Dover</name>
    <average>86</average>
  </student>
  <student index="3">
    <name>Norman Conquest</name>
    <average>85.33333333333333</average>
  </student>
</exams>
```

This approach to creating elements and attributes requires that we know the names of the elements and attributes statically, which means as literal text in the program.

Sometimes we want to create elements and attributes whose names are not known until runtime and are calculated based on information found in the source document.

Element and Attribute Constructors

XQuery has constructors for building elements and attributes whose names can be derived dynamically.

Constructors

element literalName { contents of element }

element { computedName } { contents of element }

attribute literalName { attribute value }

attribute { computedName } { attribute value }

The constructors with the literal names can replace the placement of start and end tags in an XQuery program.

The code below is another version of the body of the previous solution to Problem 8.

File: p8fb.xq

```
element exams
{
    for $s at $p in doc("roster.xml")/roster/students/student
    let $avg := myfun:average($s/exams/exam)
    order by substring-after($s/name, ' ') descending
    return element student
            {
                attribute index { $p },
                $s/name,
                element average { $avg }
            }
}
```

The results using this body will be identical to those from the previous solution with the *index* attribute.

Creating Elements and Attributes Dynamically

To illustrate the constructors for elements and attribute, we build an XML document that has entirely new attributes and elements that depend on the original document.

For testing purposes, we start with the same XML document we used in XSLT. It contains multiple elements and attributes.

File: dynamic.xml

```
<?xml version="1.0"?>
<!-- dynamic.xml -->
<root>
  <items>
    <item position="01" code="a25">
      <id>FX483</id>
      <name>Element1</name>
      <description>Debris</description>
    </item>
    <item position="02" code="b38">
      <id>FH390</id>
      <name>Element2</name>
      <description>Junk</description>
    </item>
    <item position="03" code="a88">
      <id>FA881</id>
      <name>Element3</name>
      <description>Trash</description>
    </item>
  </items>
</root>
```

In the first example we build a new XML document with elements that parallel *root*, *items*, and *item*, but the content of the *item* elements will be replaced by their attributes as new content.

Building Attributes

Observe that in this XQuery program, we assume that we do not know the names of the attributes for the *item* elements, but we can still form them into new elements.

File: da.xq

```
<newroot>
    <newitems>
    {
        for $item in doc("dynamic.xml")/root/items/item
        return <newitem>
                {
                    for $at in $item/@*
                    return element {$at/name()} { data($at) }
                }
                </newitem>
    }
    </newitems>
</newroot>
```

Results from da.xq

```
<?xml version="1.0" encoding="UTF-8"?>
<newroot>
  <newitems>
    <newitem>
      <position>01</position>
      <code>a25</code>
    </newitem>
    <newitem>
      <position>02</position>
      <code>b38</code>
    </newitem>
    <newitem>
      <position>03</position>
      <code>a88</code>
    </newitem>
  </newitems>
</newroot>
```

Building Elements and Attributes

In this XQuery program we move the *item* attributes to the content of the *newitem* element again, but this time we turn the orginal element content of the *item* elements into attributes for the *newitem* elements.

Now we have to use an element constructor to build the *newitem* elements.

File: dea.xq

```
<newroot>
   <newitems>
   {
      for $item in doc("dynamic.xml")/root/items/item
      return element newitem
               {
                    for $e in $item/*
                    return attribute { $e/name() } { $e/text() },
                    for $at in $item/@*
                    return element { $at/name() } { string($at) }
               }
   }
   </newitems>
</newroot>
```

Results from dea.xq

```
<?xml version="1.0" encoding="UTF-8"?>
<newroot>
  <newitems>
    <newitem id="FX483" name="Element1"
                                    description="Debris">
      <position>01</position>
      <code>a25</code>
    </newitem>
    <newitem id="FH390" name="Element2"
                                    description="Junk">
      <position>02</position>
      <code>b38</code>
    </newitem>
```

```
      <newitem id="FA881" name="Element3"
                                        description="Trash">
        <position>03</position>
        <code>a88</code>
      </newitem>
    </newitems>
  </newroot>
```

Quantified Expressions

Two expressions in Xquery implement the quantifiers from predicate logic: *every* and *some*.

Each takes a variables and two expressions as arguments and produces a boolean value.

These quantified expressions are evaluated in the same way they are read.

some *var* in *seq-expr* satisfies *bool-expr*

returns true if and only if there exists a value in the sequence that makes the boolean expression true.

every *var* in *seq-expr* satisfies *bool-expr*

returns true if and only if every value in the sequence makes the boolean expression true.

Both can be defined in terms of other Xquery expressions.

some *var* in *seq-expr* satisfies *bool-expr*

has the same meaning as

exists(for *var* in *seq-expr* where *bool-expr* return 1)

every *var* in *seq-expr* satisfies *bool-expr*

has the same meaning as

empty(for *var* in *seq-expr* where not(*bool-expr*) return 1)

Simple Examples

Expression	Value
every $n in 1 to 10 satisfies $n gt 0	true
every $n in 1 to 10 satisfies $n mod 3 eq 0	false
some $n in 1 to 10 satisfies $n mod 3 eq 0	true
some $n in 1 to 10 satisfies $n lt 0	false

Problem

Find out whether or not anyone scored higher than 95 on any exam in the XML document *roster.xml*.

This problem can be reworded: Tell whether there exists an exam score higher than 95 in the student information.

File: p95.xq

```
declare variable $d := doc("roster.xml");
<answer>
{
    if (some $e in
            $d/roster/students/student/exams/exam
                    satisfies number($e) gt 95)
       then "Someone scored higher than 95 on an exam."
       else "No one scored higher than 95 on an exam."
}
</answer>
```

Results from p95.xq

```
<?xml version="1.0" encoding="UTF-8"?>
<answer>Someone scored higher than 95 on
                                        an exam.</answer>
```

Problem

Find all pairs of students where the first student scored higher than the second student on each of the exams.

The idea with the solution to this problem is to generate all pairs of students (in both orders), and for each pair verify that the two students have the same number of exams and that every exam for the first student is greater than the corresponding exam of the second student.

File: higher.xq

```
declare variable $doc := doc("roster.xml");
<answer>
{
    for $s1 in $doc/roster/students/student
    for $s2 in $doc/roster/students/student
    where not($s1 is $s2)              (: want all distinct pairs :)
    return
        let $e1 := $s1/exams/exam
        let $e2 := $s2/exams/exam
        where
            count($e1) = count($e2)
            and
            (every $p in 1 to count($e1)
                satisfies number($e1[$p])>number($e2[$p]))
        return
            <pair>
            {  ($s1/name, $s2/name)  }
            </pair>
}
</answer>
```

Note: It was necessary to place the *every* expression inside parentheses.

Results from higher.xq

```
<?xml version="1.0" encoding="UTF-8"?>
<answer>
  <pair>
    <name>Guy Wire</name>
    <name>Rusty Nail</name>
  </pair>
  <pair>
    <name>Guy Wire</name>
    <name>Barb Wire</name>
  </pair>
  <pair>
    <name>Norman Conquest</name>
    <name>Rusty Nail</name>
  </pair>
  <pair>
    <name>Norman Conquest</name>
    <name>Barb Wire</name>
  </pair>
  <pair>
    <name>Eileen Dover</name>
    <name>Rusty Nail</name>
  </pair>
  <pair>
    <name>Eileen Dover</name>
    <name>Barb Wire</name>
  </pair>
</answer>
```

Problem: Prime Numbers

Write an Xquery function that tests whether a number is prime or not.

Use this function to find all the prime numbers up to 100

An integer is prime if it is greater than 1 and has the property that its only divisors are 1 and itself.

The number 2 is the only even prime.

A number n > 2 is prime if it has no divisors in the interval $2 \le d < n$. This property is equivalent to the following conditions:

- no divisors in the interval $2 \le d \le n/2$
- no divisors in the interval $2 \le d \le sqrt(n)$

Since Xquery has no square root function, we use the first condition.

File: primes.xq

```
declare namespace myfun = "myfun.slonnegr.cs.uiowa.edu";
declare function myfun:prime($n)
{
    $n = 2 or
    ($n > 2 and
      (every $d in 2 to $n idiv 2 satisfies $n mod $d > 0))
};
<primes>
{
    for $k in (1 to 100)
    where myfun:prime($k)
    return
        <prime>{ $k }</prime>
}
</primes>
```

Result from primes.xq

```
<?xml version="1.0" encoding="UTF-8"?>
<primes>
  <prime>2</prime>
  <prime>3</prime>
  <prime>5</prime>
  <prime>7</prime>
  <prime>11</prime>
  <prime>13</prime>
  <prime>17</prime>
  <prime>19</prime>
  <prime>23</prime>
  <prime>29</prime>
  <prime>31</prime>
  <prime>37</prime>
  <prime>41</prime>
  <prime>43</prime>
  <prime>47</prime>
  <prime>53</prime>
  <prime>59</prime>
  <prime>61</prime>
  <prime>67</prime>
  <prime>71</prime>
  <prime>73</prime>
  <prime>79</prime>
  <prime>83</prime>
  <prime>89</prime>
  <prime>97</prime>
</primes>
```

Joins

The join is a principal operation in the world of relational databases.

The idea is to meld the information stored in two or more database tables by comparing values in specific columns in such a way to define those rows of information that are desired.

Joins can be performed in Xquery to combine the information in two or more XML documents, selecting only that information that satisfies certain conditions on the elements in the documents.

In the following examples we consider various problems that can be solved using join operations that combine the information in *roster.xml* and a new version of the phone XML document.

File: phone.xml

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE phoneNumbers SYSTEM "phone.dtd">
<phoneNumbers>
  <title>Phone Numbers</title>
  <entries>
   <entry>
     <name gender="male">Rusty Nail</name>
     <phone>335-0055</phone>
     <city>Iowa City</city>
   </entry>
   <entry>
     <name gender="male">Guy Wire</name>
     <phone>354-9876</phone>
     <city>Coralville</city>
   </entry>
```

```
    <entry>
      <name gender="female">Eileen Dover</name>
      <phone>354-9876</phone>
      <city>Coralville</city>
    </entry>
    <entry>
      <name gender="female">Candy Barr</name>
      <phone>335-4582</phone>
      <city>North Liberty</city>
    </entry>
    <entry>
      <name gender="female">Barb Wire</name>
      <phone>337-5967</phone>
      <city>Coralville</city>
    </entry>
  </entries>
</phoneNumbers>
```

Cartesian Product

The most general sort of join includes all pairs of items from two sequences derived from XML documents.

$$AxB = \{ (a,b) \mid a \in A \text{ and } b \in B \}$$

Problem

Find all pairs of persons from the XML documents *roster.xml* and *phone.xml*.

File: cartesian.xq

```
declare variable $roster := doc("roster.xml");
declare variable $phone := doc("phone.xml");
<pairs>
{
   for $s in $roster/roster/students/student
   for $e in $phone/phoneNumbers/entries/entry
   return
      <pair>
      { concat("(", $s/name/text(), ", ", $ e/name/text(), ")") }
      </pair>
}
</pairs>
```

Results from cartesian.xq

```
<?xml version="1.0" encoding="UTF-8"?>
<pairs>
   <pair>(Rusty Nail, Rusty Nail)</pair>
   <pair>(Rusty Nail, Guy Wire)</pair>
   <pair>(Rusty Nail, Eileen Dover)</pair>
   <pair>(Rusty Nail, Candy Barr)</pair>
   <pair>(Rusty Nail, Barb Wire)</pair>
   <pair>(Guy Wire, Rusty Nail)</pair>
   <pair>(Guy Wire, Guy Wire)</pair>
   <pair>(Guy Wire, Eileen Dover)</pair>
   <pair>(Guy Wire, Candy Barr)</pair>
   <pair>(Guy Wire, Barb Wire)</pair>
   <pair>(Norman Conquest, Rusty Nail)</pair>
   <pair>(Norman Conquest, Guy Wire)</pair>
   <pair>(Norman Conquest, Eileen Dover)</pair>
   <pair>(Norman Conquest, Candy Barr)</pair>
   <pair>(Norman Conquest, Barb Wire)</pair>
   <pair>(Eileen Dover, Rusty Nail)</pair>
   <pair>(Eileen Dover, Guy Wire)</pair>
```

```
    <pair>(Eileen Dover, Eileen Dover)</pair>
    <pair>(Eileen Dover, Candy Barr)</pair>
    <pair>(Eileen Dover, Barb Wire)</pair>
    <pair>(Barb Wire, Rusty Nail)</pair>
    <pair>(Barb Wire, Guy Wire)</pair>
    <pair>(Barb Wire, Eileen Dover)</pair>
    <pair>(Barb Wire, Candy Barr)</pair>
    <pair>(Barb Wire, Barb Wire)</pair>
</pairs>
```

Observe that each document contains information for five persons. The cartesian product therefore contains 25 ordered pairs.

Inner Joins

Usually we do not want all possible ordered pairs of elements from the two sequences.

With an inner join we select only those pairs that are equal or are related in some particular way.

Problem

Find all students in *roster.xml* who are in the phone document as well.

File: equi-join.xq

```
<students>
{
    for $s in doc("roster.xml")/roster/students/student
    for $e in doc("phone.xml")/phoneNumbers/entries/entry
    where $s/name = $e/name
    return
      <student>
        { $s/name }
      </student>
}
</students>
```

Results from equi-join.xq

```
<?xml version="1.0" encoding="UTF-8"?>
<students>
  <student>
    <name>Rusty Nail</name>
  </student>
  <student>
    <name>Guy Wire</name>
  </student>
  <student>
    <name>Eileen Dover</name>
  </student>
  <student>
    <name>Barb Wire</name>
  </student>
</students>
```

More complicated conditions on the elements of the two sequences to be combined can be used to solve more challenging problems.

Problem

Find all students who are in the roster document and the phone document and are females from Coralville, showing their names and id numbers.

File: coralville.xq

```
<students>
{
   for $s in doc("roster.xml")/roster/students/student
   for $e in doc("phone.xml")/phoneNumbers/entries/entry
   where  $s/name=$e/name and
          $e/name/@gender="female" and
          $e/city="Coralville"
   return
      <student>
      { $s/@id, $s/name }
      </student>
}
</students>
```

Results from coralville.xq

```
<?xml version="1.0" encoding="UTF-8"?>
<students>
  <student id="49194">
    <name>Eileen Dover</name>
  </student>
  <student id="137745">
    <name>Barb Wire</name>
  </student>
</students>
```

Problem

Find all female students in *roster.xml* whose projects scores are all greater than 20.

File: project20.xq

```
<students>
{
   for $s in doc("roster.xml")/roster/students/student
   for $e in doc("phone.xml")/phoneNumbers/entries/entry
   where $s/name=$e/name and
            $e/name/@gender="female" and
            (every $p in $s/projects/project
                                    satisfies number($p) gt 20)
   return
      <student>
      { $s/name }
      </student>
}
</students>
```

Results from project20.xq

```
<?xml version="1.0" encoding="UTF-8"?>
<students>
  <student>
    <name>Eileen Dover</name>
  </student>
  <student>
    <name>Barb Wire</name>
  </student>
</students>
```

Problem

Find the phone number(s) of the student(s) who scored highest on exam 3.

In the solution to this problem we need to allow for the case where there are no exams in the sequence defined by the XPath expression.

To make this query more challenging we have changed the third exam score of Rusty Nail to 89 so that two students have the maximum score.

File: max3.xq

```
declare variable $roster := doc("roster.xml");
declare variable $phone := doc("phone.xml");
<students>
{
   let $exams :=
         (for $s in $roster/roster/students/student
          for $e in $phone/phoneNumbers/entries/entry
          where $s/name=$e/name
          return $s)/exams/exam[3]
   return
      if (count($exams) > gt 0)
      then
         let $mx := max($exams)
         return
            for $s in $roster/roster/students/student
            for $e in $phone/phoneNumbers/entries/entry
            where $s/name=$e/name and
                  xs:integer($s/exams/exam[3])
                                    =xs:integer($mx)
```

```
            return
               <student>
               { $e/phone }
              </student>
        else ()
}
</students>
```

Results from max3.xq

```
<?xml version="1.0" encoding="UTF-8"?>
<students>
  <student>
    <phone>335-0055</phone>
  </student>
  <student>
    <phone>354-9876</phone>
  </student>
</students>
```

Self-Joins

The sequences in a join may come from the same XML document creating what is know as a self-join.

Problem

For each person in the *phone.xml* document, find all others who live in the same city.

File: samecity.xq

```
declare variable $phone := doc("phone.xml");
<people>
{
   for $p1 in $phone/phoneNumbers/entries/entry
   return
      <person>
         <name>{ $p1/name/text() }</name>
         <neighbors>
         {
            for $p2 in $phone/phoneNumbers/entries/entry
            where not($p1 is $p2) and $p1/city=$p2/city
            return
            <name>{ $p2/name/text() }</name>
         }
         </neighbors>
      </person>
}
</people>
```

Results from samecity.xq

```
<?xml version="1.0" encoding="UTF-8"?>
<people>
   <person>
      <name>Rusty Nail</name>
      <neighbors/>
   </person>
   <person>
      <name>Guy Wire</name>
      <neighbors>
         <name>Eileen Dover</name>
         <name>Barb Wire</name>
      </neighbors>
   </person>
```

```
    <person>
      <name>Eileen Dover</name>
      <neighbors>
        <name>Guy Wire</name>
        <name>Barb Wire</name>
      </neighbors>
    </person>
    <person>
      <name>Candy Barr</name>
      <neighbors/>
    </person>
    <person>
      <name>Barb Wire</name>
      <neighbors>
        <name>Guy Wire</name>
        <name>Eileen Dover</name>
      </neighbors>
    </person>
</people>
```

Problem

For each student in *roster.xml*, find all other students who have higher scores on project 1 and on project 2.

File: higherp.xq

```
declare variable $roster := doc("roster.xml");
<students>
{
   for $s1 in $roster/roster/students/student
   return
      <student>
      {
         $s1/name,
         <higher>
         {
            for $s2 in $roster/roster/students/student
            where
               $s2/projects/project[1] gt $s1/projects/project[1]
               and
               $s2/projects/project[2] gt $s1/projects/project[2]
            return
               $s2/name
         }
         </higher>
      }
      </student>
}
</students>
```

Results from higherp.xq

```
<?xml version="1.0" encoding="UTF-8"?>
<students>
   <student>
      <name>Rusty Nail</name>
      <higher>
         <name>Barb Wire</name>
      </higher>
   </student>
```

```
    <student>
      <name>Guy Wire</name>
      <higher>
        <name>Rusty Nail</name>
        <name>Norman Conquest</name>
        <name>Barb Wire</name>
      </higher>
    </student>
    <student>
      <name>Norman Conquest</name>
      <higher>
        <name>Barb Wire</name>
      </higher>
    </student>
    <student>
      <name>Eileen Dover</name>
      <higher>
        <name>Rusty Nail</name>
        <name>Norman Conquest</name>
        <name>Barb Wire</name>
      </higher>
    </student>
    <student>
      <name>Barb Wire</name>
      <higher/>
    </student>
</students>
```

Imperative Programming: Day of the Week

The Schillo algorithm for calculating the day of the week from the month, day, and year can be expressed as an XQuery function.

File: dow.xq

```
declare namespace myfun = "myfun.slonnegr.cs.uiowa.edu";
declare function myfun:dow($m, $d, $y)
{
   let $mn := if ($m > 2) then $m - 2 else $m + 10
   let $yr := if ($m > 2) then $y else $y - 1
   let $ct := $yr idiv 100
   let $an := $yr mod 100
   let $base := (13 * $mn - 1) idiv 5 + $an idiv 4 + $ct idiv 4
   let $rem := ($base + $an + $d - 2 * $ct) mod 7
   let $offset := if ($rem < 0) then $rem + 7 else $rem
   return  if ($offset = 0) then "Sunday"
           else if ($offset = 1) then "Monday"
           else if ($offset = 2) then "Tuesday"
           else if ($offset = 3) then "Wednesday"
           else if ($offset = 4) then "Thursday"
           else if ($offset = 5) then "Friday"
           else if ($offset = 6) then "Saturday"
           else "error"
};

<dow>
{
  element last   { myfun:dow(12, 8, 2006) },
  element ww1  { myfun:dow(11, 11, 1918) },
  element ww2  { myfun:dow(12, 7, 1941) },
  element ny     { myfun:dow(9, 11, 2001) },
  element first   { myfun:dow(1, 1, 2001) }
}
</dow>
```

Results from dow.xq

```
<?xml version="1.0" encoding="UTF-8"?>
<dow>
  <last>Friday</last>
  <ww1>Monday</ww1>
  <ww2>Sunday</ww2>
  <ny>Tuesday</ny>
  <first>Monday</first>
</dow>
```

Square Roots with Newton's Method

XQuery has no function for computing the square root of a floating-point number.

We provide such a function using Newton's method.

First we show a method in Java that solves the problem.

Rather than testing for convergence, we simple calculate the first 20 terms in the sequence that converges to the square root.

Sqrt in Java

```
static double sqrt(double x)
{
    double oldx, newx = x/2.0;
    for (int k=1; k<=20; k++)
    {
        oldx = newx;
        newx = (oldx * oldx + x)/(2.0 * oldx);
    }
    return newx;
}
```

Since XQuery has no imperative variables, this algorithm needs to be expressed using recursion so that we can pass the changing values as parameters.

First we write a tail recursive version of the algorithm in Java.

This approach requires two methods, one to start the recursion and another to do the recursion, passing the changing "variable" values as parameters.

```
static double sqrt(double x)
{
    return step(20, x, x/2.0);
}

static double step(int k, double x, double oldx)
{
    double newx = (oldx * oldx + x)/(2.0 * oldx);
    if (k <= 0)
        return newx;
    else
        return step(k-1, x, newx);
}
```

The method *step* is tail recursive because the only time it calls itself is as the last action in its body.

Sqrt in XQuery

The last Java definition translates into XQuery in a straightforward manner.

Since we want to perform the calculations using the *xs:double* type, we need to express literals using *e* or *E*.

Numeric literals with a decimal point are viewed as *xs:decimal*, a type that will not be implemented as efficiently for arithmetic.

File: sqrt.xq

```
declare namespace myfun = "myfun.slonnegr.cs.uiowa.edu";

declare function myfun:sqrt($x)
{
    myfun:step(20, $x, $x div 2e1)
};

declare function myfun:step($k, $x, $oldx)
{
    let $newx := ($oldx * $oldx + $x) div ($oldx + $oldx)
    return if ($k <= 0)
           then $newx
           else myfun:step($k - 1, $x, $newx)
};

<squareRoots>
{
    for $x in (1 to 15)
    return  <root num="{ $x }">
              { myfun:sqrt($x) }
            </root>
}
</squareRoots>
```

Results from sqrt.xq

```
<?xml version="1.0" encoding="UTF-8"?>
<squareRoots>
  <root num="1">1</root>
  <root num="2">1.414213562373095</root>
  <root num="3">1.732050807568774</root>
  <root num="4">2</root>
  <root num="5">2.23606797749979</root>
  <root num="6">2.449489742783178 3</root>
  <root num="7">2.645751311064590 7</root>
  <root num="8">2.828427124746 19</root>
  <root num="9">3</root>
  <root num="10">3.162277660168379 5</root>
  <root num="11">3.316624790355 4</root>
  <root num="12">3.464101615137 7544</root>
  <root num="13">3.605551275463 989</root>
  <root num="14">3.741657386773 9413</root>
  <root num="15">3.872983346207 4166</root>
</squareRoots>
```

Static Typing and Dynamic Typing

A programming language is typed statically if variables and functions are declared to be certain types in the text of the program and the compiler verifies the consistent use of these variables and functions.

A language is dynamically typed if variables and functions are not typed in the program, so that the runtime system needs to determine the type of expressions during the execution of the program.

Strong Typing in XQuery

XQuery allows a programmer to type variables when they first occur in a program and the results of functions when they are defined.

Variables are created in a *for* clause, in a *let* clause, a *declare variable* clause, and as parameters to functions.

A variable is declared as a certain type by adding a phrase, *as some-type*, after the variable

Typing is illustrated by new versions of the day of the week problem and the square root problem.

In both cases, the results are identical to the previous versions.

The first program chooses the day string differently.

File: dowt.xq

```
declare function myfun:dow( $m as xs:integer,
                            $d as xs:integer,
                            $y as xs:integer) as xs:string
{
   let $mn as xs:integer := if ($m > 2) then $m - 2 else $m + 10
   let $yr as xs:integer := if ($m > 2) then $y else $y - 1
   let $ct as xs:integer := $yr idiv 100
   let $an as xs:integer := $yr mod 100
   let $base as xs:integer :=
                    (13 * $mn - 1) idiv 5 + $an idiv 4 + $ct idiv 4
   let $rem as xs:integer := ($base + $an + $d - 2 * $ct) mod 7
   let $offset as xs:integer :=
                    if ($rem < 0) then $rem + 7 else $rem
   return let $days := ("Sunday", "Monday", "Tuesday",
          "Wednesday","Thursday", "Friday", "Saturday")
           return $days[$offset + 1]
};
```

Square Root Program

File: sqrtt.xq

```
declare namespace myfun = "myfun.slonnegr.cs.uiowa.edu";

declare function myfun:sqrt($x as xs:double) as xs:double
{
   myfun:step(20, $x, $x div 2e1)
};

declare function myfun:step($n as xs:integer,
                            $x as xs:double,
                            $oldx as xs:double) as xs:double
{
   let $newx as xs:double :=
                       ($oldx * $oldx + $x) div ($oldx + $oldx)
   return if ($n <= 0)
               then $newx
               else myfun:step($n - 1, $x, $newx)
};
<squareRoots>
{
   for $x as xs:integer in (1 to 15)  (: coerced to xs:double :)
   return <root num="{ $x }">
               { myfun:sqrt($x) }
          </root>
}
</squareRoots>
```

Notes on Typing

- Although an XQuery implementation is not required to provide static type checking, with these declarations, static type checking is possible and dynamic type checking will be more effective.

- As with any strongly typed programming language, the XQuery processor uses the redundant information shown in the type declarations to recognize programmer errors.
- Observe the use of XML Schema types in XQuery. All of these types are available, although XQuery programmers normally use only *xs:boolean*, *xs:string*, *xs:date*, *xs:time*, *xs:dateTime*, *xs:double*, *xs:float*, *xs:decimal*, *xs:integer*, and *xs:QName*.
- Note that we are using the prefix *xs* without a namespace declaration. XQuery executes in an environment with five predefined namespaces:

Prefix	Namespace
xs	http://www.w3.org/2001/XMLSchema
fn	http://www.w3.org/2003/11/xpath-functions
xml	http://www.w3.org/XML/1998/namespace
xdt	http://www.w3.org/2003/11/xpath-datatypes
xsi	http://www.w3.org/2001/XMLSchema-instance

  The function namespace is also the default namespace, so predefined functions may be access in two ways: *fn:concat* or just *concat*.

XQuery as a Programming Language

XQuery is a fully Turing complete programming language, but there are many algorithms for which it is not well suited.

Some Language Properties

- Can be statically typed
- Can be strongly typed
- Parameters passed by value
- Static scoping is followed
- Parameters use positional correspondence
- Methods have no side effects
- Referentially transparent
- Declarative

Example: Translating to German

This is another problem that we solved in the XSLT chapter.

The goal is to translate the *phoneA.xml* document into another XML document with the same content and structure but with all of the tags expressed in German.

Since the structure of the document is known ahead of time, the XQuery program follows that structure using literal elements (start and end tags).

But since the *gender* attribute may or may not occur in a *name* element, it must be created dynamically. Other solutions may be possible.

Whether to include the *gender* attribute, the *middle* element, and the *city* element depends on their existence in the original XML document. We use the conditional expression (if-then-else) to determine whether they are present.

Observe the use of parentheses and a comma to build a single value for the first *return* command.

File: german.xq

```
<Telefonnummern>
{
   let $doc := doc("phoneA.xml")
   return
   (
      <Titel>{  $doc/phoneNumbers/title/text()  }</Titel>,
      <Eintraege>
      {
         for $e in $doc/phoneNumbers/entries/entry
         let $gender := $e/name/@gender
         let $middle := $e/name/middle/text()
         let $city := $e/city/text()
```

```
            return
            <Eintrag>
               <Name>
                  {
                     if ($gender)
                           then attribute Geschlect { $gender } else ()
                  }
                  <Vorname1>{ $e/name/first/text() }</Vorname1>
                  {
                     if ($middle)
                        then <Vorname2>{ $middle }</Vorname2>
                        else ()
                  }
                  <Nachname>{ $e/name/last/text() }
                                                   </Nachname>
               </Name>
               <Telefonnummer>
                  {  $e/phone/text()  }
               </Telefonnummer>
               {
                  if ($city) then <Stadt>{ $city }</Stadt> else ()
               }
            </Eintrag>
         }
         </Eintraege>
      )
   }
   </Telefonnummern>
```

Recall that $e/name/last produces the element *<last>Nail</last>*, whereas $e/name/last/text() produces just the string "Nail".

The output from executing this XQuery program is identical to that obtained from the stylesheet in the XSLT chapter.

You might want to compare the XQuery program with the XSLT stylesheet to form an opinion about the effectiveness of these two technologies.

XQuery in Java

No standard has been developed for executing XQuery in Java at this time.

However, several systems that link XQuery to Java have been proposed and developed.

We consider one that appears to be fairly complete (and free): Saxon.

Classes and Interfaces

The following classes and interfaces are used to run XQuery from Java in Saxon.
Some of these items come from the Java API.

net.sf.saxon.Configuration

This class holds details of user-selected configuration options.

We need to create one object of this class, but do not need any of its instance methods at this time.

```
Configuation config = new Configuration();
```

net.sf.saxon.query.StaticQueryContext

An object of this class holds information about the static context that is used when a query is compiled.

We need to create one object of this class using the Configuration object.

```
StaticQueryContext sqc =
                        new StaticQueryContext(config);
```

This object can be used to build an object that encapsulates a stream connected to an XML document.

```
DocumentInfo doc =
      sqc.buildDocument(
                  new StreamSource("phoneA.xml"));
```

net.sf.saxon.query.DynamicQueryContext

An object of this class contains a dynamic context for query execution.

We need to create one object of this class using the Configuration object.

```
DynamicQueryContext dqc =
                        new DynamicQueryContext(config);
```

This object can hold a reference to a stream that encapsulates the XML document to be queried.

```
dqc.setContextItem(
          sqc.buildDocument(
               new StreamSource("phoneA.xml")));
```

net.sf.saxon.query.XQueryExpression

An object of this class represents a compiled XQuery query. It is created by calling an overloading instance method on the StaticQueryContext object.

```
XQueryExpression exp1 =
                        sqc.compileQuery(queryString);
XQueryExpression exp2 =
                        sqc.compileQuery(readerObject);
```

XQueryExpression contains several instance methods for evaluating a compiled XQuery query.

java.util.List evaluate(DynamicQueryContent dqc)

The *evaluate* method executes the compiled query using the DynamicQueryContext object and returning a the result as a sequence (a node set).

Object evaluateSingle(DynamicQueryContent dqc)

The *evaluateSingle* method executes the compiled query, returning the first (or only) item in the result.

This method is particularly useful when the expression returns only a single value, such as a string, a number, or a boolean.

The resulting value may need to be downcast to String, Long or Double, or Boolean.

void run(DynamicQueryContext dqc,
javax.xml.transform.Result res,
java.util.Properties props)

The *run* method executes the compiled query, sending the results directly to a Result object, usually a StreamResult object from the package *javax.xml.transform.stream*.

The class StreamResult implements the interface Result and has constructors that take a Writer (a FileWriter usually), a File object, or an OutputStream (System.out usually) as a parameter.

The third parameter is a java.util.Properties object that describes various constraints on the result, such as the following.

```
props.setProperty(OutputKeys.METHOD, "xml");
props.setProperty(OutputKeys.INDENT, "yes");
```

net.sf.saxon.om.SequenceIterator
iterator(DynamicQueryContext dqc)

The *iterator* method executes the compiled query, returning an iterator object containing the node set result. See the documentation to learn how to use a SequenceIterator object since it is slightly different from java.util.Iterator.

net.sf.saxon.trans.XPathException

This exception may be thrown by *buildDocument*, *compileQuery*, *evaluate*, *evaluateSingle*, *run*, and *iterator*

javax.xml.transform.OutputKeys

This class contains a collection of useful constants (static final variables), including METHOD, INDENT, ENCODING, STANDALONE, and DOCTYPE_PUBLIC.

javax.xml.transform.stream.StreamSource

The objects of this class encapsulate an input stream of XML markup data. It has constructors that take a Reader, an InputStream, a File, or a String (a URI) as a parameter.

javax.xml.transform.stream.StreamResult

The objects of this class encapsulate an output stream of XML markup data. It has constructors that take a Writer, an OutputStream, a File, or a String (a URI) as a parameter.

Example Applications

Now we write several Java programs that use the Saxon API to execute XQuery queries to solve various problems, some of which were solved using DOM, SAX, or both.

The first example provides an XQuery evaluator that takes a string specifying the output method ("xml", "html", or "text") and the name of a file that contains an XQuery query.

Sample Execution

```
java Execute xml german.xq
java Execute html table.xq
```

where *table.xq* is a query that builds an HTML table containing the information in *phoneA.xml*.

File: Execute.java

```
import net.sf.saxon.Configuration;
import net.sf.saxon.query.DynamicQueryContext;
import net.sf.saxon.query.StaticQueryContext;
import net.sf.saxon.query.XQueryExpression;
import net.sf.saxon.trans.XPathException;

import javax.xml.transform.OutputKeys;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;

import java.io.*;
import java.util.Properties;

public class Execute
{
   public static void main(String [] args)
   {
      if (args.length != 2)
         throw new IllegalArgumentException(
                     "Usage: java Execute method queryFile");

      String meth = args[0];
      String xqFile = args[1];

      if (!meth.equals("xml") &&
            !meth.equals("html") &&
                  !meth.equals("text"))
         throw new IllegalArgumentException(
                                 "Bad method specification.");
      try
      {
         Configuration config = new Configuration();
         StaticQueryContext sqc =
                           new StaticQueryContext(config);
```

```
            DynamicQueryContext dqc =
                                new DynamicQueryContext(config);
            XQueryExpression exp =
                    sqc.compileQuery(new FileReader(xqFile));
            Properties props = new Properties();
            props.setProperty(OutputKeys.METHOD, meth);
            if (meth.equals("html"))
               props.setProperty(OutputKeys.DOCTYPE_PUBLIC,
                            "-//W3C//DTD HTML 4.01 Transitional//EN");
            int pos = xqFile.indexOf(".");
            if (pos >= 0) xqFile = xqFile.substring(0, pos);
            File file = new File(xqFile + "." + meth);
            exp.run(dqc, new StreamResult(file), props);
         }
         catch (XPathException e)
         {  System.out.println(e);  }
         catch (IOException e)
         {  System.out.println(e);  }
      }
}
```

Testing Execute.java

The XML example using *german.xq* works fine with Execute, creating a file named *german.xml*.

For HTML we have a query file *table.xq* that analyzes the XML document *phoneA.xml* and generates an HTML document that creates a table to display the information.

The query file is shown on the next page.

Observe that we need to run the *gender* attribute value through the *string* function to get the query to work properly. Otherwise the *gender* attribute is added to the *td* element in the HTML.

File: table.xq

```
<html>
   <head>
      <title>A list of phone numbers</title>
   </head>
   <body>
      <h1>Phone Numbers</h1>
      <table border="2" cellpadding="5">
         <tr bgcolor="ffaacc">
            <th>Name</th> <th>Gender</th>
            <th>Phone</th> <th>City</th>
         </tr>
         {
            for $e in
                  doc("phoneA.xml")/phoneNumbers/entries/entry
            let $first := $e/name/first/text()
            let $last := $e/name/last/text()
            let $phone := $e/phone/text()
            let $city := $e/city/text()
            order by $e/name/last
            return
               <tr>
                  <td> { concat($first," ",$last) } </td>
                  <td> { string($e/name/@gender) } </td>
                  <td> { $phone } </td>
                  <td> { $city } </td>
               </tr>
         }
      </table>
   </body>
</html>
```

Result: table.html

```
<!DOCTYPE html
          PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
     <meta http-equiv="Content-Type" content="text/html;
                                                charset=UTF-8">
    <title>A list of phone numbers</title>
  </head>
  <body>
    <h1>Phone Numbers</h1>
    <table border="2" cellpadding="5">
     <tr bgcolor="ffaacc">
        <th>Name</th>
        <th>Gender</th>
        <th>Phone</th>
        <th>City</th>
      </tr>
      <tr>
        <td>Helen Back</td>
        <td>female</td>
        <td>337-5967</td>
        <td></td>
      </tr>
      <tr>
        <td>Justin Case</td>
        <td>male</td>
        <td>354-9876</td>
        <td>Coralville</td>
      </tr>
      <tr>
        <td>Pearl Gates</td>
        <td>female</td>
        <td>335-4582</td>
        <td>North Liberty</td>
      </tr>
```

```
      <tr>
        <td>Rusty Nail</td>
        <td></td>
        <td>335-0055</td>
        <td>Iowa City</td>
      </tr>
    </table>
  </body>
</html>
```

Producing Text Output

To illustrate the use of "text" as the output method, we write a query that extracts information from *phoneA.xml* and prints it as labeled text.

File: phoneText.xq

```
    for $e in doc("phoneA.xml")/phoneNumbers/entries/entry
    let $cr := "
"
    let $g := $e/name/@gender
    let $gender := if ($g) then concat("Gender = ", $g, $cr)
                           else ()
    let $c := $e/city/text()
    let $city := if ($c) then concat("City = ", $c, $cr)
                         else ()
    order by $e/name/last
    return
      concat ( "Name = ", $e/name/first/text(), " ",
                                  $e/name/last/text(), $cr,
               $gender,
               "Phone = ", $e/phone/text(), $cr,
               $city, $cr
             )
```

Resulting text: phoneText.text

Name = Helen Back
Gender = female
Phone = 337-5967

Name = Justin Case
Gender = male
Phone = 354-9876
City = Coralville

Name = Pearl Gates
Gender = female
Phone = 335-4582
City = North Liberty

Name = Rusty Nail
Phone = 335-0055
City = Iowa City

Observations

- The extra space in front of the last three Name labels is inexplicable.
- The *string* function was not needed for the attribute value unlike the previous example.
- Escaped line feed and return characters (\n and \r) are viewed as plain text and not the control characters we want. That is why the variable *$cr* is defined as it is.
- Conclusion: XQuery is not as well suited to create text output as it is to build XML and HTML documents.

Building Java Objects from XML

In the next example we extract information from *phoneA.xml* and use it to build an ArrayList of Entry objects.

This problem was solved previously using both DOM (PhoneParser.java) and SAX (SaxPhone.java).

The result of the query that determines the number of entry elements in the XML document must cast to Long.

For some reason, we need to apply the *string* function to the results of each of the queries that extract phone data. Using *text()* did not work.

Queries for elements or attributes that do not exist produce empty strings.

This program uses the same version of the Entry class and the Name class as was used in the DOM chapter.

The execution of this program creates the same output as the program in the DOM chapter.

File: QueryPhone.java

```
import net.sf.saxon.Configuration;
import net.sf.saxon.query.DynamicQueryContext;
import net.sf.saxon.query.StaticQueryContext;
import net.sf.saxon.query.XQueryExpression;
import net.sf.saxon.trans.XPathException;

import javax.xml.transform.stream.StreamSource;
import java.util.*;

public class QueryPhone
{
   public static void main(String [] args)
   {
      List<Entry> entries = new ArrayList<Entry>();
      try
      {
         Configuration config = new Configuration();
         StaticQueryContext sqc =
                        new StaticQueryContext(config);
         DynamicQueryContext dqc =
                        new DynamicQueryContext(config);
         dqc.setContextItem(
                  sqc.buildDocument(
                           new StreamSource("phoneA.xml")));

         String query = "count(/phoneNumbers/entries/entry)";
         XQueryExpression exp = sqc.compileQuery(query);
         Long count = (Long)exp.evaluateSingle(dqc);
         int num = count.intValue();
```

```
        for (int k = 1; k <= num; k++)
        {
            query =
    "string(/phoneNumbers/entries/entry["+k+"]/name/first)";
            exp = sqc.compileQuery(query);
            String first = (String)exp.evaluateSingle(dqc);

            query =
    "string(/phoneNumbers/entries/entry["+k+"]/name/middle)";
            exp = sqc.compileQuery(query);
            String middle = (String)exp.evaluateSingle(dqc);

            query =
    "string(/phoneNumbers/entries/entry["+k+"]/name/last)";
            exp = sqc.compileQuery(query);
            String last = (String)exp.evaluateSingle(dqc);

            Name name = new Name(first, middle, last);

            query =
  "string(/phoneNumbers/entries/entry["+k+"]/name/@gender)";
            exp = sqc.compileQuery(query);
            String gender = (String)exp.evaluateSingle(dqc);

            query =
    "string(/phoneNumbers/entries/entry["+k+"]/phone)";
            exp = sqc.compileQuery(query);
            String phone = (String)exp.evaluateSingle(dqc);

            query =
    "string(/phoneNumbers/entries/entry["+k+"]/city)";
            exp = sqc.compileQuery(query);
            String city = (String)exp.evaluateSingle(dqc);
```

```
            Entry ent = new Entry(name, gender, phone, city);
            entries.add(ent);
         }
      }
      catch (XPathException ex)
      { System.out.println(ex); }

      for (int k = 0; k < entries.size(); k++)
      {
         Entry anEntry = entries.get(k);
         System.out.println(anEntry);
      }
   }
}
```

Creating an XML Document

This program builds an XML document from an ArrayList of Entry objects in the same way as the program XMLBuilder in the DOM chapter.

The idea of the program is to create an XQuery query string based on the information found in the Entry objects in an ArrayList.

A method *makeQuery* builds the query string in a StringBuffer, converts the StringBuffer to a String, and returns the String.

That query string is then executed using the Saxon Java API.

File: QueryBuilder.java

```
import net.sf.saxon.Configuration;
import net.sf.saxon.query.DynamicQueryContext;
import net.sf.saxon.query.StaticQueryContext;
import net.sf.saxon.query.XQueryExpression;
import net.sf.saxon.trans.XPathException;

import javax.xml.transform.OutputKeys;
import javax.xml.transform.stream.StreamResult;
```

```
import java.io.*;
import java.util.*;

public class QueryBuilder
{
   static String mkQuery(List<Entry> entries)
   {
      StringBuffer query =
                        new StringBuffer("<phoneNumbers>");
      query.append("{ ( element title { \"Phone Numbers\" }, ");
      query.append("element entries { (");
      boolean first = true;
      for (Entry anEntry : entries)
      {
         if (!first) query.append(", ");        // put a comma
         first = false;                         // between entries
         query.append("element entry { ( element name { (");
         Name name = anEntry.getName();
         String gender = anEntry.getGender();
         if (gender != null)
            query.append("attribute gender {\"" + gender + "\" },");
         query.append("element first { \"" + name.getFirst() + "\" },");
         String middle = name.getMiddle();
         if (middle != null)
            query.append("element middle {\"" + middle + "\" },");
         query.append("element last { \""
                                        + name.getLast() + "\" } ) },");
         query.append("element phone { \""
                                   + anEntry.getPhone() + "\" }");
         String city = anEntry.getCity();
         if (city != null && !city.equals(""))
            query.append(", element city { \"" + city + "\" }");
         query.append(") }");
      }
```

```
        query.append(") } ) } </phoneNumbers>");
        return query.toString();
    }

    public static void main(String [] args)
    {
        List<Entry> entries = new ArrayList<Entry>();
        entries.add(new Entry(new Name("Robin", "Banks"),
                                    "354-4455"));
        entries.add(new Entry(new Name("Forrest", "Murmers"),
                                    "male", "341-6152", "Solon"));
        entries.add(new Entry(new Name("Barb", "A", "Wire"),
                                    "337-8182", "Hills"));
        entries.add(new Entry(new Name("Isabel", "Ringing"),
                                    "female", "335-5985", null));
        String query = mkQuery(entries);
        System.out.println(query);     // for debugging query
        try
        {
            Configuration config = new Configuration();
            StaticQueryContext sqc =
                        new StaticQueryContext(config);
            DynamicQueryContext dqc =
                        new DynamicQueryContext(config);
            XQueryExpression exp = sqc.compileQuery(query);
            Properties props = new Properties();
            props.setProperty(OutputKeys.METHOD, "xml");
            File file = new File("newPhone.xml");
            exp.run(dqc, new StreamResult(file), props);
        }
        catch (Exception e)
        { System.out.println(e); }
    }
}
```

Query String Built by QueryBuilder

```
<phoneNumbers>
{
   (  element title { "Phone Numbers" },
      element entries
      {  (element entry
         { ( element name
               { (element first { "Robin" },
                  element last { "Banks" } ) },
             element phone { "354-4455" }) },
         element entry { ( element name
               { (attribute gender {"male" },
                  element first { "Forrest" },
                  element last { "Murmers" } ) },
               element phone { "341-6152" },
               element city { "Solon" }) },
         element entry
         { ( element name
               { (element first { "Barb" },
                  element middle {"A" },
                  element last { "Wire" } ) },
               element phone { "337-8182" },
               element city { "Hills" }) },
          element entry
         { ( element name
               { (attribute gender {"female" },
                  element first { "Isabel" },
                  element last { "Ringing" } ) },
             element phone { "335-5985" }) }) } )
}
</phoneNumbers>
```

File: newPhone.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<phoneNumbers>
  <title>Phone Numbers</title>
  <entries>
    <entry>
      <name>
        <first>Robin</first>
        <last>Banks</last>
      </name>
      <phone>354-4455</phone>
    </entry>
    <entry>
      <name gender="male">
        <first>Forrest</first>
        <last>Murmers</last>
      </name>
      <phone>341-6152</phone>
      <city>Solon</city>
    </entry>
    <entry>
      <name>
        <first>Barb</first>
        <middle>A</middle>
        <last>Wire</last>
      </name>
      <phone>337-8182</phone>
      <city>Hills</city>
    </entry>
    <entry>
      <name gender="female">
        <first>Isabel</first>
        <last>Ringing</last>
      </name>
      <phone>335-5985</phone>
    </entry>
  </entries>
</phoneNumbers>
```