XQuery Processor

CSE 636 Fall 2008 Project Phase 1

November 7, 2008

The course project is the construction of an XQuery processor for a subset of the XPath and XQuery languages as described below. In Phase 1, you will construct a naive query processor that receives an XQuery expression, parses it into an abstract tree representation and evaluates the expression using a recursive evaluation routine that given an XQuery expression (path, concatenation, element creation, etc) and one or more input XML document nodes it produces a list of output nodes.

The project consists of three major components:

1. **Parser:** Use the XPath and XQuery grammars provided below to construct an XQuery parser using the JavaCC parser generator.
2. **XPath:** Implement the XPath processor according to the semantics given in Section 1.
3. **XQuery:** Implement the XQuery processor, by utilizing the XPath processor, according to the semantics given in Section 2.

You may team up with at most one partner for the project.

1 The XPath Sub-Language of XQuery

We consider XPath, the sublanguage of XQuery which deals with specifying paths along which the XML tree is to be navigated to extract data from it. For the sake of simplicity, we will only consider a restriction of the full W3C XPath standard.

Any expression generated by the following context-free grammar is a valid XPath expression:

$$
\begin{array}{rrcl}
\text{(absolute path)} & ap & \rightarrow & \mathsf{doc}(\mathit{fileName})/rp \\
 & & \mid & \mathsf{doc}(\mathit{fileName})//rp \\
 & & & \\
\text{(relative path)} & rp & \rightarrow & \mathit{tagName} \mid * \mid . \mid .. \mid \mathsf{text}() \\
 & & \mid & (rp) \mid rp_1/rp_2 \mid rp_1//rp_2 \mid rp[f] \mid \cancel{rp_1, rp_2}(rp_1, rp_2) \\
 & & & \\
\text{(path filter)} & f & \rightarrow & rp \mid rp_1 = rp_2 \mid rp_1 \ \mathsf{eq}\ rp_2 \mid rp_1 == rp_2 \mid rp_1 \ \mathsf{is}\ rp_2 \\
 & & \mid & (f) \mid f_1 \ \mathsf{and}\ f_2 \mid f_1 \ \mathsf{or}\ f_2 \mid \mathsf{not}\ f
\end{array}
$$

The above grammar only helps us check whether an XPath expression p has the correct syntax. But what is its **meaning**, that is, what is the result of extracting from an XML tree the data reachable by navigating along p? To answer this question, we need to settle the following problem. How can one define the meaning of any XPath expression without explicitly listing each such expression and, for each possible XML document, the associated result? This would be an unfeasible approach, as there are infinitely many XPath expressions, as well as infinitely many XML trees.

The solution is a standard one, adopted from programming language theory. We will define a function which, applied to any XPath expression p and XML tree rooted at node n, will return the list of nodes reachable by navigating along p. Recall that we consider two kinds of nodes in the XML tree: element nodes, and text nodes. Text nodes may be associated to element nodes.

We will use the following functions:

function	returns
$[\![ap]\!]_A$	the list of (element or text) nodes reached by navigating from the root along absolute path ap
$[\![rp]\!]_R(n)$	the list of (element or text) nodes reachable from element node n by navigating along the path specified by relative XPath expression rp.
$[\![f]\!]_F(n)$	true if and only if the filter f holds at node n
$\mathsf{root}(fn)$	the root of the XML tree corresponding to the document fn
$\mathsf{children}(n)$	the list of children of element node n, ordered according to the document order
$\mathsf{parent}(n)$	a singleton list containing the parent of element node n, if n has a parent. The empty list otherwise.
$\mathsf{tag}(n)$	the tag labeling element node n
$\mathsf{txt}(n)$	the text node associated to element node n

List manipulations We will also use the following notation on list manipulations. $< a, b, c >$ denotes a list of three entries (a is the first, c the last). $<>$ denotes the empty list, and $< e >$ is the singleton list with unique entry e.

In the following, l_1, l_2 are the lists $l_1 =< x_1, \dots, x_n >$ and $l_2 =< y_1, \dots, y_m >$.

$$l_1, l_2$$

denotes the concatenation of the two lists, i.e. the list $< x_1, \dots, x_n, y_1, \dots, y_m >$.

$$\mathsf{unique}(l_1)$$

denotes the list obtained by scanning l from head to tail and removing any duplicate elements that have been previously encountered.

For example, $< 1, 2, 3 >, < 2, 3, 4 >=< 1, 2, 3, 2, 3, 4 >$, and $\mathsf{unique}(< 1, 2, 3 >, < 2, 3, 4 >) =< 1, 2, 3, 4 >$.

The notation $< f(x) \mid x \leftarrow l_1 >$ is called a list comprehension, and it is shorthand for a loop which binds variable x in order against the entries of l_1, and returns the list with entries given by applying f to each binding of x:

$$< f(x) \mid x \leftarrow l_1 >=< f(x_1), \dots, f(x_n) >$$

A list comprehension can have arbitrarily many condition and variable binding expressions. In general, if $c(v_1, \dots, v_k)$ is a condition involving variables v_1 through v_k,

$$< f(v_1, \dots, v_k) \mid v_1 \leftarrow l_1, \dots, v_k \leftarrow l_k, c(v_1, v_2, \dots, v_k) >$$

is short for the function defined by the following pseudocode fragment:

```
result := <>
foreach v1 in l1
 ...
   foreach vk in lk
     if c(v1,...,vk) then
       result := result, <f(v1,...,vk)>
return result
```

We are now ready to define the meaning of an XPath expression:

$$\llbracket \mathsf{doc}(\mathit{fileName})/rp \rrbracket_A = \llbracket rp \rrbracket_R(\mathsf{root}(\mathit{fileName})) \tag{1}$$
$$\llbracket \mathsf{doc}(\mathit{fileName})//rp \rrbracket_A = \llbracket .//rp \rrbracket_R(\mathsf{root}(\mathit{fileName})) \tag{2}$$

$$\llbracket \mathit{tagName} \rrbracket_R(n) = < x \mid x \leftarrow \llbracket * \rrbracket_R(n), \mathsf{tag}(x) = \mathit{tagName} > \tag{3}$$
$$\llbracket * \rrbracket_R(n) = \mathsf{children(n)} \tag{4}$$
$$\llbracket . \rrbracket_R(n) = < n > \tag{5}$$
$$\llbracket .. \rrbracket_R(n) = \mathsf{parent}(n) \tag{6}$$
$$\llbracket \mathsf{text()} \rrbracket_R(n) = \mathsf{txt}(n) \tag{7}$$
$$\llbracket (rp) \rrbracket_R(n) = \llbracket rp \rrbracket_R(n) \tag{8}$$
$$\llbracket rp_1/rp_2 \rrbracket_R(n) = \mathsf{unique}(< y \mid x \leftarrow \llbracket rp_1 \rrbracket_R(n), y \leftarrow \llbracket rp_2 \rrbracket_R(x) >) \tag{9}$$
$$\llbracket rp_1//rp_2 \rrbracket_R(n) = \mathsf{unique}(\llbracket rp_1/rp_2 \rrbracket_R(n), \llbracket rp_1/*//rp_2 \rrbracket_R(n)) \tag{10}$$
$$\llbracket rp[f] \rrbracket_R(n) = < x \mid x \leftarrow \llbracket rp \rrbracket_R(n), \llbracket f \rrbracket_F(x) > \tag{11}$$
$$\cancel{\llbracket rp_1, rp_2 \rrbracket_R}\llbracket (rp_1, rp_2) \rrbracket_R(n) = \llbracket rp_1 \rrbracket_R(n), \llbracket rp_2 \rrbracket_R(n) \tag{12}$$

$$\llbracket rp \rrbracket_F(n) = \llbracket rp \rrbracket_R(n) \neq <> \tag{13}$$
$$\llbracket rp_1 = rp_2 \rrbracket_F(n) = \llbracket rp_1 \; \mathsf{eq} \; rp_2 \rrbracket_F(n) = \exists x \in \llbracket rp_1 \rrbracket_R(n) \; \exists y \in \llbracket rp_2 \rrbracket_R(n) \; x \; \mathsf{eq} \; y \tag{14}$$
$$\llbracket rp_1 == rp_2 \rrbracket_F(n) = \llbracket rp_1 \; \mathsf{is} \; rp_2 \rrbracket_F(n) = \exists x \in \llbracket rp_1 \rrbracket_R(n) \; \exists y \in \llbracket rp_2 \rrbracket_R(n) \; x \; \mathsf{is} \; y \tag{15}$$
$$\llbracket (f) \rrbracket_F(n) = \llbracket f \rrbracket_F(n) \tag{16}$$
$$\llbracket f_1 \; \mathsf{and} \; f_2 \rrbracket_F(n) = \llbracket f_1 \rrbracket_F(n) \wedge \llbracket f_2 \rrbracket_F(n) \tag{17}$$
$$\llbracket f_1 \; \mathsf{or} \; f_2 \rrbracket_F(n) = \llbracket f_1 \rrbracket_F(n) \vee \llbracket f_2 \rrbracket(n) \tag{18}$$
$$\llbracket \mathsf{not} \; f \rrbracket_F(n) = \neg \llbracket f \rrbracket_F(n) \tag{19}$$

Value-based and Identity-based Equality XPath distinguishes among two types of equality. Two XML nodes n and m are value-equal (denoted n eq m or $n = m$) if and only if the trees rooted at them are isomorphic. That is, if

- $\mathsf{tag}(n) = \mathsf{tag}(m)$ and
- $\mathsf{text}(n) = \mathsf{text}(m)$ and
- n has as many children as m and
- for each k, the k^{th} child of n and the k^{th} child of m are value-equal.

In other words, n is a copy of m. n and m are id-equal (denoted n is m or $n == m$) if and only if they are identical. That is, a node n is only id-equal to itself. n is not id-equal to a distinct copy of itself. Note that id-equality implies value-equality, but not vice versa.

2 The XQuery Sub-Language for the Project

The W3C XQuery standard contains many bells and whistles which we will abstract from for the sake of simplicity. For our purposes, the syntax of XQuery is defined as follows:

$$
\begin{array}{lrcl}
\text{(XQuery)} & XQ & \rightarrow & Var \mid StringConstant \mid ap \\
 & & \mid & (XQ) \mid \cancel{XQ_1, XQ_2}(XQ_1, XQ_2) \mid XQ/rp \mid XQ//rp \\
 & & \mid & \langle tagName\rangle\{XQ\}\langle/tagName\rangle \\
 & & \mid & forClause\ whereClause\ returnClause \\
 & forClause & \rightarrow & \mathsf{for}\ Var_1\ \mathsf{in}\ XQ_1, Var_2\ \mathsf{in}\ XQ_2, \ldots, Var_n\ \mathsf{in}\ XQ_n \\
 & whereClause & \rightarrow & \epsilon \mid \mathsf{where}\ Cond \\
 & returnClause & \rightarrow & \mathsf{return}\ XQ \\
 & Cond & \rightarrow & XQ_1 = XQ_2 \mid XQ_1\ \mathsf{eq}\ XQ_2 \\
 & & \mid & XQ_1 == XQ_2 \mid XQ_1\ \mathsf{is}\ XQ_2 \\
 & & \mid & (Cond) \mid Cond_1\ \mathsf{and}\ Cond_2 \mid Cond_1\ \mathsf{or}\ Cond_2 \mid \mathsf{not}\ Cond
\end{array}
$$

Element and Text Node Constructors We will use the function

$$\mathsf{makeElem}(t, l)$$

which takes as arguments a tag name t and a (potentially empty) list of XML nodes l and returns a new XML element node n with $\mathsf{tag}(n) = t$ and $\mathsf{children}(n) = l$. Similarly,

$$\mathsf{makeText}(s)$$

takes as argument a string constant s and returns an XML text node with value s.

$$
\begin{array}{rcll}
[\![Var]\!]_X & = & < Var > & (20) \\
[\![StringConstant]\!]_X & = & < \mathsf{makeText}(StringConstant) > & (21) \\
[\![ap]\!]_X & = & [\![ap]\!]_A & (22) \\
[\![(XQ)]\!]_X & = & [\![XQ]\!]_X & (23) \\
\cancel{[\![XQ_1, XQ_2]\!]_X}[\![(XQ_1, XQ_2)]\!]_X & = & [\![XQ_1]\!]_X, [\![XQ_2]\!]_X & (24) \\
[\![XQ/rp]\!]_X & = & \mathsf{unique}(< m \mid n \leftarrow [\![XQ]\!]_X, m \leftarrow [\![rp]\!]_R(n) >) & (25) \\
[\![XQ//rp]\!]_X & = & \mathsf{unique}([\![XQ/rp_2]\!]_X, [\![XQ/*//rp_2]\!]_X) & (26) \\
[\![\langle tagName\rangle\{XQ\}\langle/tagName\rangle]\!]_X & = & < \mathsf{makeElem}(tagName, [\![XQ]\!]_X) > & (27)
\end{array}
$$

$$
\begin{array}{rcll}
[\![XQ_1\ \mathsf{eq}\ XQ_2]\!]_C = [\![XQ_1 = XQ_2]\!]_C & = & \exists x \in [\![XQ_1]\!]_X\ \exists y \in [\![XQ_2]\!]_X\ x\ \mathsf{eq}\ y & (28) \\
[\![XQ_1\ \mathsf{is}\ XQ_2]\!]_C = [\![XQ_1 == XQ_2]\!]_C & = & \exists x \in [\![XQ_1]\!]_X\ \exists y \in [\![XQ_2]\!]_X\ x\ \mathsf{is}\ y & (29) \\
[\![(Cond)]\!]_C & = & [\![Cond]\!]_C & (30) \\
[\![Cond_1\ \mathsf{and}\ Cond_2]\!]_C & = & [\![Cond_1]\!]_C \wedge [\![Cond_2]\!]_C & (31) \\
[\![Cond_1\ \mathsf{or}\ Cond_2]\!]_C & = & [\![Cond_1]\!]_C \vee [\![Cond_2]\!]_C & (32) \\
[\![\mathsf{not}\ Cond]\!]_C & = & \neg[\![Cond]\!]_C & (33)
\end{array}
$$

Finally, we have

$$
\left[\!\!\left[\begin{array}{ll}
\mathsf{for} & Var_1\ \mathsf{in}\ XQ_1, \ldots, \\
 & Var_n\ \mathsf{in}\ XQ_n \\
\mathsf{where} & Cond \\
\mathsf{return} & XQ_{n+1}
\end{array}\right]\!\!\right]_X
= \begin{array}{ll}
< & [\![XQ_{n+1}]\!]_X \mid \\
 & Var_1 \leftarrow [\![XQ_1]\!]_X, \ldots, Var_n \leftarrow [\![XQ_n]\!]_X \\
 & [\![Cond]\!]_C \quad >
\end{array} \qquad (34)
$$

3 Deliverables

Component\Due Date	Command Line	Output Format
Parser **Monday, October 6th**	XQueryParser xqueryfile.xq	Your XQueryParser class should print out the abstract tree of the input XQuery expression.
XPath Processor **Friday, October 24th**	XPathProcessor xpathfile.xq	Your XPathProcessor class should print out the result of evaluating the XPath expression in the input file.
XQuery Processor **Monday, November 10th**	XQueryProcessor xqueryfile.xq	Your XQueryProcessor class should print out the result of evaluating the XQuery expression in the input file.

For each deliverable, **only one** of the team members should email the instructor a .zip archive containing the following items:

- A README.txt file describing the archive contents.
- A PDF file describing your implementation.
- All the source code you wrote for the deliverable in folder **src**.
- All the compiled code for the deliverable in folder **bin**.

Important All Java classes mentioned in the "Command Line" column above should be located within the **bin** folder. Since your deliverables will be graded using a script, all trees that are printed out should follow the same format and tabulation as the output format of DOMPrinter class provided with this specification.