XQuery Semantics

CSE 232B

April 26, 2007

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.[1]

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

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

The above grammar only helps us check whether an XPath expression p has the correct syntax. But what is its **meaning**, i.e. 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? Note that 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

[1] For the sake of simplicity, we will only consider a restriction of the full W3C XPath standard.

function	returns
$\llbracket ap \rrbracket_A$	the list of (element or text) nodes reached by navigating from the root along absolute path ap
$\llbracket rp \rrbracket_R(n)$	the list of (element or text) nodes reachable from element node n by navigating along the path specified by relative XPath expression rp.
$\llbracket f \rrbracket_F(n)$	true if and only if the filter f holds at node n
$\mathsf{root}(\mathit{fn})$	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.

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

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

$$\llbracket \mathit{tagName} \rrbracket_R(n) = < c \mid x \leftarrow \llbracket * \rrbracket_R(n), \mathsf{tag}(n) = \mathit{tagName} > \tag{3}$$

$$\llbracket * \rrbracket_R(n) = \mathsf{children(n)} \tag{4}$$

$$\llbracket . \rrbracket_R(n) = < n > \tag{5}$$

$$\llbracket .. \rrbracket_R(n) = \mathsf{parent}(n) \tag{6}$$

$$\llbracket \mathsf{text()} \rrbracket_R(n) = \mathsf{txt}(n) \tag{7}$$

$$\llbracket (rp) \rrbracket_R(n) = \llbracket rp \rrbracket_R(n) \tag{8}$$

$$\llbracket rp_1/rp_2 \rrbracket_R(n) = \mathsf{unique}(< y \mid x \leftarrow \llbracket rp_1 \rrbracket_R(n), y \leftarrow \llbracket rp_2 \rrbracket_R(x) >) \tag{9}$$

$$\llbracket rp_1//rp_2 \rrbracket_R(n) = \mathsf{unique}(\llbracket rp_1/rp_2 \rrbracket_R(n), \llbracket rp_1/*//rp_2 \rrbracket_R(n)) \tag{10}$$

$$\llbracket rp[f] \rrbracket_R(n) = < x \mid x \leftarrow \llbracket rp \rrbracket_R(n), \llbracket f \rrbracket_F(x) > \tag{11}$$

$$\llbracket rp_1, rp_2 \rrbracket_R(n) = \llbracket rp_1 \rrbracket_R(n), \llbracket rp_2 \rrbracket_R(n) \tag{12}$$

$$\llbracket rp \rrbracket_F(n) = \llbracket rp \rrbracket_R(n) \neq <> \tag{13}$$

$$\llbracket rp_1 = rp_2 \rrbracket_F(n) = \llbracket rp_1 \ \mathsf{eq}\ rp_2 \rrbracket_F(n) = \exists x \in \llbracket rp_1 \rrbracket_R(n)\ \exists y \in \llbracket rp_2 \rrbracket_R(n)\ x\ \mathsf{eq}\ y \tag{14}$$

$$\llbracket rp_1 == rp_2 \rrbracket_F(n) = \llbracket rp_1 \ \mathsf{is}\ rp_2 \rrbracket_F(n) = \exists x \in \llbracket rp_1 \rrbracket_R(n)\ \exists y \in \llbracket rp_2 \rrbracket_R(n)\ x\ \mathsf{is}\ y \tag{15}$$

$$\llbracket (f) \rrbracket_F(n) = \llbracket f \rrbracket_F(n) \tag{16}$$

$$\llbracket f_1 \ \mathsf{and}\ f_2 \rrbracket_F(n) = \llbracket f_1 \rrbracket_F(n) \wedge \llbracket f_2 \rrbracket_F(n) \tag{17}$$

$$\llbracket f_1 \ \mathsf{or}\ f_2 \rrbracket_F(n) = \llbracket f_1 \rrbracket_F(n) \vee \llbracket f_2 \rrbracket(n) \tag{18}$$

$$\llbracket \mathsf{not}\ f \rrbracket_F(n) = \neg \llbracket f \rrbracket_F(n) \tag{19}$$

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 viceversa.

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:

$$
\begin{array}{llcl}
\text{(XQuery)} & XQ & \rightarrow & Var \mid StringConstant \mid ap \\
 & & \mid & (XQ_1) \mid XQ_1, XQ_2 \mid XQ_1/rp \\
 & & \mid & \langle tagName \rangle \{XQ_1\} \langle / tagName \rangle \\
 & & \mid & forClause\ letClause\ whereClause\ returnClause \\
 & & \mid & letClause\ XQ_1 \\
 & & & \\
 & forClause & \rightarrow & \mathsf{for}\ Var_1\ \mathsf{in}\ XQ_1, Var_2\ \mathsf{in}\ XQ_2, \ldots, Var_n\ \mathsf{in}\ XQ_n \\
 & & & \\
 & letClause & \rightarrow & \epsilon \mid \mathsf{let}\ Var_{n+1} := XQ_{n+1}, \ldots, Var_{n+k} := XQ_{n+k} \\
 & & & \\
 & whereClause & \rightarrow & \epsilon \mid \mathsf{where}\ Cond \\
 & & & \\
 & returnClause & \rightarrow & \mathsf{return}\ XQ_1 \\
 & & & \\
 & Cond & \rightarrow & XQ_1 = XQ_2 \mid XQ_1\ \mathsf{eq}\ XQ_2 \\
 & & \mid & XQ_1 == XQ_2 \mid XQ_1\ \mathsf{is}\ XQ_2 \\
 & & \mid & \mathsf{empty}(XQ_1) \\
 & & \mid & \mathsf{some}\ Var_1\ \mathsf{in}\ XQ_1, \ldots, Var_n\ \mathsf{in}\ XQ_n\ \mathsf{satisfies}\ Cond \\
 & & \mid & (Cond_1) \mid Cond_1\ \mathsf{and}\ Cond_2 \mid Cond_1\ \mathsf{or}\ Cond_2 \mid \mathsf{not}\ Cond_1
\end{array}
$$

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.

Variable Scope As in any programming language with variables, we need to define the scope of variables. We first note that variables can be defined only by for, let and some clauses. We impose the following scoping rules, which are quite natural for any programming language with block structure.

- The scope of variables bound in a for clause extends to the corresponding (as given by production 8 of non-terminal XQ above) let clause (if any), where clause (if any) and return clause.
- The scope of the variables bound in a let clause extends to the following where and return clauses (if the applicable production is no. 8 above), or to the XQ_1 (if the applicable production is no. 9 above).
- The scope of the variables bound in a some clause extends to the condition in the satisfies clause.
- Moreover, within any for, let or some clause, every XQ_i used to bind variable Var_i may depend on the previously defined variables.

A definition of variable v will override within the definition's scope any prior definition of variable v. For instance, in a query

$$\mathsf{for}\ v\ \mathsf{in}\ XQ_1,\ w\ \mathsf{in}\ XQ_2\ \mathsf{let}\ v := XQ_3\ \mathsf{where}\ Cond\ \mathsf{return}\ XQ_4$$

any reference to v in $Cond$ and XQ_4 refers to the definition using XQ_3, while any reference in XQ_2 refers to the definition using XQ_1.

Evaluating Expressions with Free Variables in a Context Since we intend to evaluate an expression by evaluating its sub-expressions first, we need to cover the case when the sub-expression mentions free variables defined outside. To this end, we will record all variable bindings in an auxiliary data structure called a *context*, and pass the context as argument to the evaluation function, which will look up prior variable bindings in the context. Think of a contet as an associative array which relates variables to the value they are bound to. A context supports two operations:

- $\{Var \mapsto v\}C$ extends the context C with a new binding for variable *Var* to value v. This operation has no side-effect, i.e. it does not change C, instead returning a brand new context which copies from C all bindings of variables other than *Var*.
- $C(Var)$ is the operation of looking up the binding of variable Var in context C, yielding the value *Var* was bound to.[2]

To support the override rule for variable definitions, we require any context to behave as follows:

$$(\{Var \mapsto u\}C)(Var) = u$$

which implies in particular (for $C = \{Var \mapsto v\}C'$) that

$$(\{Var \mapsto u\}\{Var \mapsto v\}C')(Var) = u.$$

The Evaluation Functions The function evaluating an XQuery expression XQ within a context C is $\llbracket XQ \rrbracket_X(C)$, and it returns a list of element and text nodes. The function evaluating a condition *Cond* within a context C is $\llbracket Cond \rrbracket_{Cn}(C)$ and it evaluates to a boolean. We define the two functions below.

$$\begin{aligned}
\llbracket Var \rrbracket_X(C) &= \; < \; C(Var) \; > \qquad (20)\\
\llbracket StringConstant \rrbracket_X(C) &= \; < \mathsf{makeText}(StringConstant) > \qquad (21)\\
\llbracket ap \rrbracket_X(C) &= \llbracket ap \rrbracket_A \qquad (22)\\
\llbracket (XQ_1) \rrbracket_X(C) &= \llbracket XQ_1 \rrbracket_X(C) \qquad (23)\\
\llbracket XQ_1, XQ_2 \rrbracket_X(C) &= \llbracket XQ_1 \rrbracket_X(C), \llbracket XQ_2 \rrbracket_X(C) \qquad (24)\\
\llbracket XQ_1/rp \rrbracket_X(C) &= \mathsf{unique}(< m \mid n \leftarrow \llbracket XQ_1 \rrbracket_X(C), m \leftarrow \llbracket rp \rrbracket_R(n) >) \qquad (25)\\
\llbracket \langle tagName \rangle \{XQ_1\} \langle / tagName \rangle \rrbracket_X(C) &= \; < \mathsf{makeElem}(tagName, \llbracket XQ_1 \rrbracket_X(C)) > \qquad (26)\\
\\
\llbracket XQ_1 \; \mathsf{eq} \; XQ_2 \rrbracket_{Cn}(C) = \llbracket XQ_1 = XQ_2 \rrbracket_{Cn}(C) &= \exists x \in \llbracket XQ_1 \rrbracket_X(C) \; \exists y \in \llbracket XQ_2 \rrbracket_X(C) \; x \; \mathsf{eq} \; y \qquad (27)\\
\llbracket XQ_1 \; \mathsf{is} \; XQ_2 \rrbracket_{Cn}(C) = \llbracket XQ_1 == XQ_2 \rrbracket_{Cn}(C) &= \exists x \in \llbracket XQ_1 \rrbracket_X(C) \; \exists y \in \llbracket XQ_2 \rrbracket_X(C) \; x \; \mathsf{is} \; y \qquad (28)\\
\llbracket \mathsf{empty}(XQ_1) \rrbracket_{Cn}(C) &= \llbracket XQ_1 \rrbracket_X(C) = <> \qquad (29)\\
\left\llbracket \begin{array}{l} \mathsf{some} \; Var_1 \; \mathsf{in} \; XQ_1, \ldots, Var_n \; \mathsf{in} \; XQ_n \\ \mathsf{satisfies} \; Cond \end{array} \right\rrbracket_{Cn} (C) &= \exists v_1 \in \llbracket XQ_1 \rrbracket_X(C_0)\\
&\quad \ldots\\
&\quad \exists v_n \in \llbracket XQ_n \rrbracket_X(C_{n-1})\\
&\quad \llbracket Cond \rrbracket_{Cn}(C_n) \qquad (30)\\
\text{where } C_0 := C, \; C_i := \{Var_i \mapsto v_i\}C_{i-1}, \; i \in [1, \ldots, n] &
\end{aligned}$$

[2] We shall assume that variables are always defined before being used (this can be easily checked at parsing time) and therefore define the evaluation only for well-formed XQuery expressions.

$$
\begin{aligned}
[\![(Cond_1)]\!]_{Cn}(C) &= [\![Cond_1]\!]_{Cn}(C) && (31)\\
[\![Cond_1 \ \textsf{and}\ Cond_2]\!]_{Cn}(C) &= [\![Cond_1]\!]_{Cn}(C) \wedge [\![Cond_2]\!]_{Cn}(C) && (32)\\
[\![Cond_1 \ \textsf{or}\ Cond_2]\!]_{Cn}(C) &= [\![Cond_1]\!]_{Cn}(C) \vee [\![Cond_2]\!]_{Cn}(C) && (33)\\
[\![\textsf{not}\ Cond_1]\!]_{Cn}(C) &= \neg[\![Cond_1]\!]_{Cn}(C) && (34)
\end{aligned}
$$

Finally, we have

$$
\left[\!\!\left[\begin{array}{ll} \textsf{let} & Var_1 \ := \ XQ_1, \ldots, Var_n \ := \ XQ_n \\ XQ_{n+1} & \end{array}\right]\!\!\right]_X (C) \quad = \quad [\![XQ_{n+1}]\!]_X(C_n) \qquad (35)
$$

$$
\text{where } C_0 := C,\ C_i := \{Var_i \mapsto [\![XQ_i]\!]_X(C_{i-1})\}C_{i-1},\ i \in [1,\ldots,n]
$$

$$
(36)
$$

$$
\left[\!\!\left[\begin{array}{ll} \textsf{for} & Var_1 \ \textsf{in}\ XQ_1, \ldots, \\ & Var_n \ \textsf{in}\ XQ_n \\ \textsf{let} & Var_{n+1} := XQ_{n+1}, \ldots, \\ & Var_{n+k} := XQ_{n+k} \\ \textsf{where} & Cond \\ \textsf{return} & XQ_{n+k+1} \end{array}\right]\!\!\right]_X (C) \quad = \quad \begin{array}{ll} < & [\![XQ_{n+k+1}]\!]_X(C_{n+k}) \mid \\ & v_1 \leftarrow [\![XQ_1]\!]_X(C_0), \\ & \ldots, \\ & v_n \leftarrow [\![XQ_n]\!]_X(C_{n-1}), \\ & [\![Cond]\!]_{Cn}(C_{n+k}) \ > \end{array} \qquad (37)
$$

$$
\begin{aligned}
&\text{where } C_0 := C,\ C_i := \{Var_i \mapsto v_i\}C_{i-1},\ i \in [1,\ldots,n]\\
&\text{and } \ C_j := \{Var_j \mapsto [\![XQ_j]\!]_X(C_{j-1})\}C_{j-1},\ j \in [n+1,\ldots,n+k]
\end{aligned}
$$

Notice that the effect of the let construct is simply that of extending the context with bindings for the variables declred in the construct.