1 The XPath Sub-language of XQuery

We consider XPath, the sub-language of XQuery which deals with specifying paths along which the XML tree is to be navigated to extract data from it.\footnote{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.

\[
\begin{align*}
(\text{absolute path}) \quad ap & \rightarrow \text{doc(fileName)}/rp \\
& \quad \mid \text{doc(fileName)}/rp \\
(\text{relative path}) \quad rp & \rightarrow \text{tagName} \mid * \mid . \mid .. \mid \text{text}() \\
& \quad \mid (rp) \mid rp_1/rp_2 \mid rp_1//rp_2 \mid rp[f] \mid rp_1, rp_2 \\
(\text{path filter}) \quad f & \rightarrow rp \mid rp_1 = rp_2 \mid rp_1 \text{ eq } rp_2 \mid rp_1 \equiv rp_2 \mid rp_1 \text{ is } rp_2 \\
& \quad \mid (f) \mid f_1 \text{ and } f_2 \mid f_1 \text{ or } f_2 \mid \text{ not } f
\end{align*}
\]
function \returns \[
\text{the list of (element or text) nodes reached by navigating from the root along absolute path} \ ap
\]
\[
\text{the list of (element or text) nodes reachable from element node} \ n \ \text{by navigating along the path specified by relative XPath expression} \ rp.
\]
\[
\text{true if and only if the filter} \ f \ \text{holds at node} \ n
\]
\[
\text{the root of the XML tree corresponding to the document} \ fn
\]
\[
\text{the list of children of element node} \ n, \text{ordered according to the document order}
\]
\[
\text{a singleton list containing the parent of element node} \ n, \text{if} \ n \ \text{has a parent. The empty list otherwise.}
\]
\[
\text{the tag labeling element node} \ n
\]
\[
\text{the text node associated to element node} \ n
\]

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

In the following, \(l_1, l_2\) are the lists \(< \ x_1, \ldots, x_n \ \text{> and} \ l_2 =< \ y_1, \ldots, y_m \ \text{>.}\)

\[
l_1, l_2
\]

\text{denotes the concatenation of the two lists, i.e. the list} \ < \ x_1, \ldots, x_n, y_1, \ldots, y_m \ \text{>}

\[
\text{unique}(l_1)
\]

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

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

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 \(x\):

\[
< f(x) \mid x \leftarrow l_1 >= < f(x_1), \ldots, f(x_n) >
\]

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

\[
< f(v_1, \ldots, v_k) \mid v_1 \leftarrow l_1, \ldots, v_k \leftarrow l_k, c(v_1, v_2, \ldots, v_k) >=< f(v_1, \ldots, v_k) >
\]

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

```plaintext
result := <>
for each v1 in l1
  ...
  for each 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.

\[
\begin{align*}
[\text{doc(fileName)}/\text{rp}]_A & = [\text{rp}]_R(\text{root(fileName)}) \\
[\text{doc(fileName)}/\text{rp}]_A & = [/\text{rp}]_R(\text{root(fileName)})
\end{align*}
\]

\[
\begin{align*}
[\text{tagName}]_R(n) & = < c | x \leftarrow [[*]]_R(n), \text{tag}(n) = \text{tagName} > \\
[[*]]_R(n) & = \text{children}(n) \\
[[.]]_R(n) & = < n > \\
[[.\.]]_R(n) & = \text{parent}(n) \\
[\text{text()}]_R(n) & = \text{txt}(n) \\
[[\text{rp}]]_R(n) & = [[\text{rp}]]_R(n) \\
[rp_1 / rp_2]_R(n) & = \text{unique}(< y | x \leftarrow [rp_1]_R(n), y \leftarrow [rp_2]_R(x) >) \\
[rp_1 / * / rp_2]_R(n) & = \text{unique}([rp_1 / rp_2]_R(n), [rp_1 / * / rp_2]_R(n)) \\
[rp[f]]_R(n) & = < x | x \leftarrow [rp]_R(n), [f]_F(x) > \\
[rp_1, rp_2]_R(n) & = [[rp_1]_R(n), [rp_2]_R(n)
\end{align*}
\]

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

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

In other words, \( n \) is a copy of \( m \). \( n \) and \( m \) are id-equal (denoted \( n \ \text{is} \ m \) 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:
(XQuery) \[
XQ \rightarrow \text{Var} \mid \text{StringConstant} \mid \text{ap} \\
(XQ_1) \mid XQ_1, XQ_2 \mid XQ_1/rp \\
\langle \text{tagName} \rangle \{XQ_1\} \{/\text{tagName}\} \\
\text{forClause letClause whereClause returnClause}
\]

\text{forClause} \rightarrow \text{for Var}_1 \in XQ_1, \text{Var}_2 \in XQ_2, \ldots, \text{Var}_n \in XQ_n

\text{letClause} \rightarrow \epsilon \mid \text{let} \text{Var}_{n+1} := XQ_{n+1}, \ldots, \text{Var}_{n+k} := XQ_{n+k}

\text{whereClause} \rightarrow \epsilon \mid \text{where} \text{Cond}

\text{returnClause} \rightarrow \text{return} XQ_1

\text{Cond} \rightarrow XQ_1 = XQ_2 \mid XQ_1 \text{ eq } XQ_2 \\
XQ_1 == XQ_2 \mid XQ_1 \text{ is } XQ_2 \\
\text{empty}(XQ_1) \\
\text{some Var}_1 \text{ in } XQ_1, \ldots, \text{Var}_n \text{ in } XQ_n \text{ satisfies } \text{Cond} \\
(\text{Cond}_1) \mid \text{Cond}_1 \text{ and } \text{Cond}_2 \mid \text{Cond}_1 \text{ or } \text{Cond}_2 \mid \text{not } \text{Cond}_1

\textbf{Element and Text Node Constructors}  

We will use the function \(\text{makeElem}(t, l)\)

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 \(\text{tag}(n) = t\) and \(\text{children}(n) = l\). Similarly,

\(\text{makeText}(s)\)

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

\[
\begin{align*}
[\text{Var}]_X & = < \text{Var} > \quad (20) \\
[\text{StringConstant}]_X & = < \text{makeText}([\text{StringConstant}]) > \quad (21) \\
[\text{ap}]_X & = [\text{ap}]_A \quad (22) \\
[(XQ_1)]_X & = [XQ]_X \quad (23) \\
[XQ_1, XQ_2]_X & = [XQ_1]_X, [XQ_2]_X \quad (24) \\
[XQ_1/rp]_X & = [[XQ_1]_X, m \leftarrow [rp]_R(n)] \quad (25) \\
[(\text{tagName})\{XQ_1\}\{/\text{tagName}\}]_X & = < \text{makeElem}(\text{tagName}, [XQ_1]_X) > \quad (26)
\end{align*}
\]

\[
\begin{align*}
[XQ_1 \text{ eq } XQ_2]_C & = [XQ_1 = XQ_2]_C \quad = \exists x \in [XQ_1]_X \exists y \in [XQ_2]_X \ x \text{ eq } y \quad (27) \\
[XQ_1 \text{ is } XQ_2]_C & = [XQ_1 == XQ_2]_C \quad = \exists x \in [XQ_1]_X \exists y \in [XQ_2]_X \ x \text{ is } y \quad (28) \\
[\text{empty}(XQ_1)]_C & = [XQ_1]_X = \emptyset \quad (29) \\
\text{some } \text{Var}_1 \text{ in } XQ_1, \ldots, \text{Var}_n \text{ in } XQ_n \text{ satisfies } \text{Cond} & \quad \Rightarrow \exists \text{Var}_1 \in [XQ_1]_X \ldots \exists \text{Var}_n \in [XQ_n]_X \text{ Cond} \quad (30)
\end{align*}
\]

\[
\begin{align*}
[\text{Cond}_1]_C & = [\text{Cond}_1]_C \quad (31) \\
[\text{Cond}_1 \text{ and } \text{Cond}_2]_C & = [\text{Cond}_1]_C \land [\text{Cond}_2]_C \quad (32) \\
[\text{Cond}_1 \text{ or } \text{Cond}_2]_C & = [\text{Cond}_1]_C \lor [\text{Cond}_2]_C \quad (33) \\
[\text{not } \text{Cond}_1]_C & = \neg[\text{Cond}_1]_C \quad (34)
\end{align*}
\]
Finally, we have

\[
\begin{align*}
\left[ \begin{array}{l}
\text{for} & \text{Var}_1 \text{ in } Q_1, \ldots, \\
\text{Var}_n \text{ in } Q_n \\
\text{where} & \text{Cond} \\
\text{return} & Q_{n+k+1}
\end{array} \right]_X & = < [Q_{n+k+1}]_X \mid \\
& \text{Var}_1 \leftarrow [Q_1]_X, \ldots, \text{Var}_n \leftarrow [Q_n]_X \\
& \text{Cond}_C > (35)
\end{align*}
\]

\[
\left[ \begin{array}{l}
\text{for} & \text{Var}_1 \text{ in } Q_1, \ldots, \\
\text{Var}_n \text{ in } Q_n \\
\text{let} & \text{Var}_{n+1} \leftarrow Q_{n+1}, \ldots, \\
\text{Var}_{n+k} \leftarrow Q_{n+k} \\
\text{where} & \text{Cond} \\
\text{return} & Q_{n+k+1}
\end{array} \right]_X = \left[ \begin{array}{l}
\text{for} & \text{Var}_1 \text{ in } Q_1, \ldots, \\
\text{Var}_n \text{ in } Q_n \\
\text{where} & \theta \text{Cond} \\
\text{return} & \theta Q_{n+k+1}
\end{array} \right]_X (36)
\]

where \( \theta \) is the substitution \( \{ \text{Var}_{n+1} \mapsto Q_{n+1}, \ldots, \text{Var}_{n+k} \mapsto Q_{n+k} \} \). For any expression \( E \), \( \theta E \) denotes the new expression obtained by applying \( \theta \) to \( E \). Notice that the effect of the let construct is simply that of declaring the shorthands \( \text{Var}_{n+1}, \ldots, \text{Var}_{n+k} \) for the expressions \( Q_{n+1}, \ldots, Q_{n+k} \), respectively.