An Introduction to XPath Processing

Stijn Vansummeren

February 26, 2010

1 Introduction

In these notes, we present an efficient algorithm for processing (a fragment of) XPath 2.0 queries that was proposed in 2002 by Gottlob, Koch, and Pichler [3].[1] Before this proposal, it was unclear whether a truly efficient algorithm for processing XPath existed. Indeed, the XPath 1.0 and XPath 2.0 W3C recommendations [1, 2] explain the semantics of XPath location paths in a way that immediately suggests the following naive recursive algorithm for evaluating location paths. Let t be the tree representation of the input XML document, let x be the current context node, and let Q be the XPath location path that we need to evaluate. For simplicity, we write Q.head for the first location step in Q, and Q.tail for the list obtained from Q by deleting the first location step. In particular, Q.tail is empty if Q contains only one step expression.

Example 1. To illustrate, when Q = child::a[position()=1]/descendant::b then Q.head = child::a[position()=1] and Q.tail = descendant::b. Furthermore, (Q.tail).head = descendant::b and (Q.tail).tail is empty.

Algorithm 1: NAIVE(Q, t, x)

```
Input: a location path $Q$, an XML tree $t$, context node $x$
Result: set of all nodes that can be reached by $Q$ in $t$ starting form $x$
1  begin
2      Compute $S$ := set of all nodes reachable from context node $x$ in $t$ by step $Q$.head
3      if $Q.tail$ is empty then
4          return $S$
5      else
6          Initialize set $R := \emptyset$
7          foreach $y$ in $S$ do  $R := R \cup$ NAIVE($Q$.tail, $t$, $y$)
8          return $R$
9      end
10 end
```

NAIVE, shown in Algorithm 1, processes a location path step by step: it first computes the set of nodes S that are reachable from the context node x by applying the step Q.head. If there

[1]It should be noted that in [3] even more refined algorithms are given for processing XPath queries. For simplicity, we present here a relatively simple one.

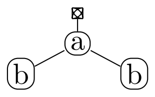
remain further steps to be evaluated (i.e. Q.tail is non-empty), then PROCESS-LOCATION-PATH is recursively called for each $y \in S$, which serves as the new context node. Otherwise, S is the result.

Remark 1. Note that the result of an XPath 2.0 location path is normally a *sequence* of nodes, sorted in document order. NAIVE, in contrast, returns a *set* of nodes, and is hence unordered. We can, however, always sort this set in document order very efficiently in $\mathcal{O}(N \log N)$ time using standard sorting techniques, where N is the number of nodes to be sorted. For this reason, all algorithms in these notes will return without loss of generality a *set* of items instead of a sequence. We implicitly assume that this set is turned into a sequence as a post-processing step.

We claim that NAIVE is inherently inefficient. To see why, we introduce the following definitions.

Definition 1. Let $\#t$ denote the number of nodes in the XML tree t and let $\#Q$ denote the number of location steps in location path Q.

Example 2. Consider the simple XML document `<a><b/><b/></a>`[2]. Its tree t representation is



and hence $\#t = 4$ (remember that each XML tree has a special root node). Moreover, the location path $Q =$ child::a[position()=1]/descendant::b from Example 1 has $\#Q = 2$ since it consists of two location steps.

The following proposition shows that NAIVE requires time exponential in the length of the XPath query in the worst case.

Proposition 1. *The worst-case running time of* NAIVE *is* $\Omega(\#t^{\frac{\#Q}{2}-1})$.

Proof. To obtain this proposition, we construct a family $\{Q_1, Q_2, Q_3, \dots\}$ of XPath location paths such that:

- every Q_i has $\#Q_i = 2i$ for $1 \leq i$;
- the time needed to evaluate NAIVE(Q_i, t, x) for arbitrary t and x is at least $(\#t)^{i-1}$.

In particular, we construct this family by induction on i as follows:

$$Q_1 = \text{ancestor-or-self::*[not(parent::*)] / descendant-or-self::*}$$
$$Q_{i+1} = \text{ancestor-or-self::*[not(parent::*)] / descendant-or-self::* / } Q_i$$

Hence, Q_1 first moves from the context node to the root node by means of the

ancestor-or-self::*[not(parent::*)]

step, and then selects all nodes by means of the descendant-or-self::* step. This switching between the root node and all its descendants is repeated i times by Q_i. So, for every odd

[2]For simplicity, we omit all XML declarations in our examples.

step in Q_i the set S computed in line 2 of Algorithm 1 contains only 1 node, while for every even step in Q_i, it contains $\#t$ nodes.

Let us analyze the time complexity of NAIVE on (Q_i, t, x) for an arbitrary i, t, and x. Suppose for simplicity that we can compute line 2 in constant time. We actually need more time to evaluate a location step (a concrete algorithm will be given in Section 4.1), but if we can show that NAIVE takes time at least $\Omega(\#t^{i-1})$ when location steps can be evaluated in *constant* time, then NAIVE will also take time at least $\Omega(\#t^{i-1})$ time when location steps require more than constant time.

Evaluating line 2 of NAIVE for the first step ancestor-or-self::*[not(parent::*)] of Q_i hence takes some constant time C. Since the root node is the only node that satisfies this step, the set S hence computed consists only of the root node. NAIVE then recursively calls itself with the root as context node. Hence, line 2 is now executed for the second step descendant-or-self::* of Q_i, costing an additional time C. The set S now consists of all nodes in the document. If $i = 1$ we are finished, otherwise we need to call evaluate Q_{i-1} for every node in S. In other words, Q_{i-1} is evaluated $\#t$ times. Hence, the time T_i needed to evaluate NAIVE on input (Q_i, t, x) is given by the recurrence

$$T_i = \begin{cases} 2C & \text{if } i = 1 \\ 2C + \#t \times T_{i-1} & \text{otherwise} \end{cases}$$

In other words,

$$T_i = 2C + 2C\#t + 2C\#t^2 + \cdots + 2C\#t^{i-1}.$$

Clearly, this is $\Omega(\#t^{i-1})$. Substituting $i = \frac{\#Q_i}{2}$, we obtain $T_i = \Omega(\#t^{\frac{\#Q_i}{2}-1})$, as desired. $\square$

Curiously enough, experiments performed by Gottlob, Koch, and Pichler with various XPath processors (including the XPath processor of Microsoft Internet Explorer 6) indicate that in 2002 many XPath processors actually used algorithms like NAIVE with inherent exponential-time behavior. You are encouraged (but not required) to read Section 2 in their paper in this respect. In contrast, in the following sections, a processing algorithm is given that runs in worst-case time polynomial in both $\#t$ and $\#Q$. It is hence vastly more efficient in the worst case.

To simplify the presentation, we will focus on a small fragment of XPath 2.0 and ignore features like external functions, generalized comparisons, namespaces, ... which can easily be added. We introduce the necessary definitions and notation in Section 2, and present the actual fragment in Section 3. Section 4 describes the algorithm.

2 The execution environment

The algorithm will work with the tree representation of an XML document that was introduced in the lesson 'HTML & XML'. In particular, recall that the tree representation of an XML document is a tree in which each node is of a particular type. We will consider here only *root, element, text*, and *attribute* nodes, and ignore *comment, namespace*, and *processing instruction* nodes (which, again can easily be added). Recall also that the root node is the only of type *root*, which is the parent of the root document element node. Only element nodes can occur as interior nodes; attribute and text nodes must always be leaves. Element nodes and attribute nodes have a name associated with them; text nodes do not. Furthermore, all nodes have a string value associated with them: for text and attribute nodes this is simply their content and value, respectively; for root and element nodes this is the concatenation of the string values of all descendant text nodes in document order.

To access the information of an XML tree t we assume given the following operations:

Operation	**Semantics**	**Complexity**
$\mathit{root}(t)$	Returns the root node of t (of type *root*)	Constant time ($\mathcal{O}(1)$)
$\mathit{firstchild}(x,t)$	Returns the first child of node x in t , *nil* if x has no children. The first child may be an attribute node, and hence needs not be an element or text node	Constant time ($\mathcal{O}(1)$)
$\mathit{parentnode}(x,t)$	Returns the parent node of x in t, *null* if x has no parent	Constant time ($\mathcal{O}(1)$)
$\mathit{nextsib}(x,t)$	Returns the next sibling node of x in t, and *null* if x has no next sibling	Constant time ($\mathcal{O}(1)$)
$\mathit{prevsib}(x,t)$	Returns the previous sibling of x in t, and *null* if x has no next sibling	Constant time ($\mathcal{O}(1)$)
$\mathit{is\_elemnode}(x,t)$	Returns true if x is an element node, false otherwise	Constant time ($\mathcal{O}(1)$)
$\mathit{is\_textnode}(x,t)$	Returns true if x is a text node, false otherwise	Constant time ($\mathcal{O}(1)$)
$\mathit{is\_attrnode}(x,t)$	Returns true if x is an attribute node, false otherwise	Constant time ($\mathcal{O}(1)$)
$\mathit{strval}(x,t)$	Returns the string value of node x in t	Linear time ($\mathcal{O}(\mathit{size}(t))$)
$\mathit{name}(x,t)$	Returns the name of node x in t. Yields a runtime error if x is not an element or attribute node	Constant time ($\mathcal{O}(1)$)

Here, $\mathit{size}(t)$ refers to the total size of tree t, defined as follows:

Definition 2. Define $\mathit{size}(t)$ to be the sum of the number of nodes $\#t$ and of the total length of all node labels, attribute values, and text content, viewed as a string.

Example 3. To illustrate, the tree t in Example 2 consists of 4 nodes. The name of each element node consists of a single character. Hence, $\mathit{size}(t) = 4 + 3 = 7$.

In other words: we assume that the data structure used to encode XML trees allows us to (1) move to the root; (2) to move from a node to its neighboring nodes; and (3) compute the string value and names of nodes in linear time. Items (1) and (2) can be realized in constant time by maintaining pointers to the root and (for each node) to neighboring nodes. Item (3) may require linear time in the total size of t since, for instance, the string value of an element node requires us to visit all of its descendant text nodes, and to concatenate all of their string contents.

In addition to the above operations on trees, we assume that we are given the following operations on sets, numbers, booleans, and strings. Let $|s|$ denote the length of a string s. We assume that numbers fit in one machine word so that operations on numbers take only constant time.

Operation	**Semantics**	**Complexity**
$\|S\|$	Returns the cardinality of set S	Constant time ($\mathcal{O}(1)$)
empty?(S)	Returns boolean true if set S is empty, and false otherwise	Constant time ($\mathcal{O}(1)$)
to_num(s)	Converts string s to a number	Linear time ($\mathcal{O}(\|s\|)$
streq(s_1, s_2)	Returns true if strings s_1 and s_2 are equal, and false otherwise	$\mathcal{O}(\|s_1\| + \|s_2\|)$
$+$,$-$,$*$,*div*	Numerical operations on numbers	Constant time ($\mathcal{O}(1)$)
numeq(s_1, s_2)	Returns true if numbers s_1 and s_2 are equal, and false otherwise	Constant time ($\mathcal{O}(1)$)

3 The fragment

In order to easily illustrate the main idea of the evaluation algorithm, we restrict ourselves to the following fragment of XPath 2.0:

- We only consider XPath expressions in abbreviated syntax. This is not a restriction, since we can always transform an abbreviated XPath expression into an unabbreviated XPath expression in a pre-processing step.
- Moreover, we ignore most of the standard functions and operators. It is straightforward to add more functions, although it should be stressed that the worst-case time complexity of our evaluation algorithm may grow if we do so. (For example, if we add support for a function that requires exponential time , then our evaluation algorithm will also be exponential in the worst case.)
- We ignore variables. Again, it is not difficult to add them.
- We ignore namespaces.
- Finally, we require that the programmer makes all typecasts that normally happen implicitly in XPath (from the empty sequence to the boolean false, from the non-empty sequence to the boolean true, from a string to a number, ...) explicit through the use the (standard) function fn:empty and number(). Other typecasts can be added without much difficulty.

  To illustrate this restriction, consider the expression child::a[descendant::text()] that selects all a-labeled children of the context node that have a text-node descendant. Since the predicate [descendant::text()] implicitly converts the secquence returned by descendant::text() into a boolean, this expression is not in our fragment. Nevertheless, it can equivalently be written as child::a[not(fn:empty(descendant::text()))], which is in our fragment.

Concretely, the XPath fragment that we consider is given by the grammar shown in Figure 1. There, terminal symbols are typeset in sans serif and non-terminals in *italics*.

There are four main kinds of expressions:

1. sequence expressions (produced by *seqexp*) that evaluate to a set of nodes (actually, a sequence of nodes, but again this sequence can be obtained from the set by sorting it in document order);

$$\begin{array}{rcl}
\mathit{seqexp} & ::= & \mathit{path} \mid /\mathit{path} \\
\mathit{path} & ::= & \mathit{step} \mid \mathit{step}/\mathit{path} \\
\mathit{step} & ::= & \mathit{axis}::\mathit{test} \mid \mathit{axis}::\mathit{test}[\mathit{boolexp}]\ldots[\mathit{boolexp}] \\
\mathit{axis} & ::= & \mathsf{self} \mid \mathsf{child} \mid \mathsf{parent} \mid \mathsf{descendant} \mid \mathsf{ancestor} \mid \mathsf{descendant\text{-}or\text{-}self} \mid \mathsf{ancestor\text{-}or\text{-}self} \\
 & \mid & \mathsf{following} \mid \mathsf{preceding} \mid \mathsf{following\text{-}sibling} \mid \mathsf{preceding\text{-}sibling} \mid \mathsf{attribute} \\
\mathit{test} & ::= & \mathsf{*} \mid \mathit{name} \mid \mathsf{element()} \mid \mathsf{text()} \\
 & & \\
\mathit{boolexp} & ::= & \mathsf{true()} \mid \mathsf{false()} \mid \mathsf{not}(\mathit{boolexp}) \\
 & \mid & \mathit{boolexp}\ \mathsf{and}\ \mathit{boolexp} \mid \mathit{boolexp}\ \mathsf{or}\ \mathit{boolexp} \\
 & \mid & \mathit{comp} \\
 & \mid & \mathsf{fn{:}empty}(\mathit{seqexp}) \\
\mathit{comp} & ::= & \mathit{strexp} \mathrel{\mathsf{=}} \mathit{strexp} \mid \mathit{numexp} \mathrel{\mathsf{=}} \mathit{numexp} \\
 & & \\
\mathit{numexp} & ::= & \mathsf{count}(\mathit{seqexp}) \mid \mathsf{position()} \mid \mathsf{last()} \mid \mathsf{number()} \mid \mathit{numliteral} \\
 & \mid & \mathit{numexp} + \mathit{numexp} \mid \mathit{numexp} - \mathit{numexp} \\
 & \mid & \mathit{numexp} * \mathit{numexp} \mid \mathit{numexp}\ \mathsf{div}\ \mathit{numexp} \\
 & & \\
\mathit{strexp} & ::= & \mathsf{string()} \mid \mathsf{string}(\mathit{seqexp}) \mid \mathsf{name()} \mid \mathsf{name}(\mathit{seqexp}) \mid \mathit{strliteral}
\end{array}$$

Figure 1: The syntax of our XPath fragment

2. boolean expression (produced by *boolexp*) that evaluate to a single boolean value;

3. numerical expressions (produced by *numexp*) that evaluate to a single (real) number;

4. string expressions (produced by *strexp*) that evaluate to a single string.

No production rules are given for the non-terminals *numliteral* and *strliteral*. We assume that these non-terminals capture numeric literals (like the constant 5, the constant 3.1415 etc) and string literals (like "test literal"), respectively.

4 The Algorithm

Intuitively, NAIVE is inefficient because it evaluates subexpressions for each context node *separately*. As such, it recomputes many intermediate results over and over again. Indeed, suppose that we evaluate a location path $step_1/step_2/\ldots/step_n$ starting from context node x. NAIVE begins by computing the set $\{x_1, \ldots, x_k\}$ of all nodes reachable from x by $step_1$. For each of $1 \leq i \leq k$, it then recursively calls itself with context node x_i. To illustrate in what sense intermediate results are recomputed many times, let R_i be the set of context nodes reachable form x_i by $step_2$. When processing $step_2/\ldots/step_n$ with x_i as context node, NAIVE recursively calls itself for each node in R_i. However, since each x_i is treated in isolation, and since R_i and R_j need not be disjoint for $i \neq j$, NAIVE ends up duplicating the recursive computation for nodes that occur in multiple of the R_i's. In particular, if node y occurs in every R_i (i.e., $y \in R_1 \cap \cdots \cap R_k$) then this computation is duplicated k times.

In essence, the algorithm introduced in this section circumvents such recomputation by not evaluating subexpressions for each context node separately. Instead, each subexpression is evaluated *for all relevant context nodes together.* In Database Systems, this is called *batch processing.*

In contrast to NAIVE, which takes as input a triple of the form (*expression, tree, single context*), our evaluation algorithm will hence take as input a triple of the form (*expression, tree, vector-of-contexts*). The vector of contexts contains all contexts for which we need to evaluate the expression. For the purpose of evaluating *numexp* expressions we will need not only the context node, but also the context position and context size. This brings us to the following definition.

Definition 3. A *context on an XML tree t* is a triple $\langle x, p, s\rangle$ such that:

- x is a node in t;
- s is a natural number called the *context size.* It must be smaller than the total number of nodes in t; and
- p is a natural number called the *context position.* It must be smaller than s.

Remark 2. In the handbook, as well as in the lesson 'Navigating XML trees with XPath' the context was defined to consist also of a set of variable bindings, a function library, and a set of namespace declarations. These components are not necessary here since we ignore variables, external functions, and namespaces.

Actually, we are not going to define a single evaluation function, but four evaluation functions, one for each of the four main kinds of expressions *seqexp*, *boolexp*, *numexp*, and *strexp*. These evaluation functions (EVAL-SEQEXP, EVAL-BOOLEXP, EVAL-NUMEXP, and EVAL-STREXP, respectively) take as input a triple $(e, t, \vec{c})$ where (1) e is an expression of the corresponding kind; (2) t is the input XML tree; and (3) $\vec{c}$ is a variable-length vector of contexts on t:

- For EVAL-SEQEXP, the result is a vector $(R_1, \ldots, R_k)$ of sets of nodes in t such that R_i is the result of evaluating the input expression e on t starting from context c_i, for each $1 \leq i \leq k$. For ease of use, we will abbreviate "set of nodes in t" simply as *nodeset* in what follows.
- For EVAL-BOOLEXP, the result is a vector $(b_1, \ldots, b_k)$ of booleans such that b_i is the result of evaluating e on t starting from context c_i.
- For EVAL-NUMEXP, the result is a vector $(n_1, \ldots, n_k)$ of numbers such that n_i is the result of evaluating e on t starting from context c_i.
- For EVAL-STREXP, the result is a vector $(s_1, \ldots, s_k)$ of strings such that s_i is the result of evaluating e on t starting from context c_i.

4.1 Evaluating *seqexp* expressions

EVAL-SEQEXP is the function that actually requires the most sophistication, the other three functions are straightforward to define.

In particular, EVAL-SEQEXP, shown in Algorithm 2, first checks whether the input expression e is an absolute location path of the form $/e'$. If so, then the result of evaluating e will be the same for each input context: it is the result of evaluating e' starting from the root. Hence,

Algorithm 2: EVAL-SEQEXP$(e, t, \vec{c})$

```
Input: a seqexp $e$; a tree $t$; and a variable-length vector of contexts $\vec{c} = (c_1, \ldots, c_k)$;
Result: a vector of node sets $(R_1, \ldots, R_k)$ of the same length as $\vec{c}$ such that $R_i$ is the
        result of evaluating $e$ on $c_i$.
1  begin
2      Let $\vec{c} = (\langle x_1, p_1, s_1\rangle, \langle x_2, p_2, s_2\rangle, \ldots, \langle x_k, p_k, s_k\rangle)$
3      if $e$ is /$e'$ then                          /* $e$ is an absolute location path */
4          Compute $R :=$ EVAL-SEQEXP$(e', t, (\langle root(t), 1, 1\rangle))$
5          return $(R, \ldots, R)$                                       /* $k$ times */
6      else                                         /* $e$ is a relative location path */
7          Let $e = step_1$ / ... / $step_n$
8          Initialize $(R_1, \ldots, R_k) := (\{x_1\}, \ldots, \{x_k\})$
9          for $i := 1$ to $n$ do
10             $(R_1, \ldots, R_k) :=$ PROCESS-STEP$(step_i, t, (R_1, \ldots, R_k))$
11         end
12         return $(R_1, \ldots, R_k)$
13 end
```

EVAL-SEQEXP recursively calls itself with the context $\langle root(t), 1, 1\rangle$, and copies the resulting node set k times to the output. If e is not absolute, but of the form $step_1$ / ... / $step_n$, it evaluates each step in turn using the auxiliary function PROCESS-STEP shown in Algorithm 3. PROCESS-STEP takes as input a triple $(step, t, \vec{X})$ where *step* is the location step to evaluate, t is an XML tree, and $\vec{X} = (X_1, \ldots, X_k)$ is a variable-length vector of node sets. It returns a vector of node sets $(R_1, \ldots, R_k)$ of the same size as $\vec{X}$ such that R_i is the set of nodes reachable from any node in X_i by following *step*:

$$R_i = \{y \mid y \text{ reachable in } t \text{ from some } x \in X_i \text{ by following } step\}.$$

In particular, PROCESS-STEP operates as follows. Assume that the input step is of the form $\alpha::\tau[e_1]\ldots[e_m]$ were α is the axis, τ is the node test, and $e_1,\ldots,$ e_m are the predicates. PROCESS-STEP first computes in line 2 the set S of pairs $\langle x, S_x\rangle$ such that $x \in X_1 \cup \cdots \cup X_k$ and y is both (1) reachable from x by axis α and (2) satisfies node test τ. (We will see how to compute S efficiently in Section 5). Note that, although a node x may occur in multiple of the sets X_i $(1 \leq i \leq k)$, the axis and test are only evaluated once per node x.

In lines 7-14, PROCESS-STEP removes form S those pairs $\langle x, y\rangle$ that do not satisfy the predicates. Since each predicate is a boolean expression (see Figure 1), PROCESS-STEP evaluates each predicate by recursively calling EVAL-BOOLEXP on a vector of contexts $\vec{c}$. This vector contains, for every $\langle x, y\rangle \in S$, a context $\langle y, p, s\rangle$ such that:

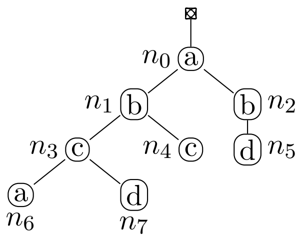
- s is the size of the nodeset $S_x = \{z \mid \langle x, z\rangle \in S\}$ that resulted from evaluating the axis and test starting from x.
- p is the position of y in this nodeset S_x if we were to sort it in document order (if α is a forward axis) or reverse document order (if α is a backward axis).

In particular, p and s are computed in line 9, where assume to have function $idx(\alpha, y, S_x)$ that computes the correct position by sorting S_x as required by the axis α. This can be done in $\mathcal{O}(N \log N)$ time using known sorting techniques, where $N = |S_x|$.

Algorithm 3: PROCESS-STEP(α::$\tau[e_1]\dots[e_m], t, \vec{X}$)

```
Input: a location step $\alpha$::$\tau[e_1]\dots[e_m]$ with axis $\alpha$, node test $\tau$ and (variable-length)
       predicates $e_1, \dots, e_n$; a tree $t$; and a variable-length vector of node sets
       $\vec{X} = (X_1, \dots, X_k)$;
Result: a vector of node sets $(R_1, \dots, R_k)$ of the same length as $\vec{X}$ such that $R_i$ is the
        set of all nodes that can be reached by the step starting from a node in $X_i$
1  begin
2      Compute
       $S := \{\langle x, y\rangle \mid x \in X_1 \cup \cdots \cup X_k, y$ reachable from $x$ by $\alpha, y$ satisfies test $\tau\}$
3      for $i := 1$ to $m$ do        /* process the predicates from left to right */
4          foreach $x$ in $X_1 \cup \cdots \cup X_k$ do Compute $S_x := \{z \mid \langle x, z\rangle \in S\}$
5          Initialize context vector $\vec{c}$ to the empty vector
6          Initialize $j := 0$
7          foreach $\langle x, y\rangle$ in $S$ do
8              $j := j + 1$
9              Add context $\langle y, idx(\alpha, y, S_x), |S_x|\rangle$ to $\vec{c}$
10             Set $j_{\langle x,y\rangle} := j$;
11         end
12         Let $(b_1, \dots, b_l) :=$ EVAL-BOOLEXP$(e_i, t, \vec{c})$
13         Remove from $S$ all $\langle x, y\rangle$ for which $b_{j_{\langle x,y\rangle}} = \mathsf{false}$
14     end
15     for $i := 1$ to $k$ do
16         Compute $R_i := \{y \mid \langle x, y\rangle \in S, x \in X_i\}$
17     end
18     return $(R_1, \dots, R_k)$
19 end
```

Example 4. To illustrate the operation of PROCESS-STEP, assume that *step* is the location step $\mathsf{child} :: * [\mathsf{position}() = 2]$ and that t is the following tree.



Assume further that we call PROCESS-STEP on $(step, t, \vec{X})$ where $\vec{X} = (\{n_1, n_2\}, \{n_2\})$. So, $k = 2$. Then:

- In line 2 we compute $S = \{\langle n_1, n_3\rangle, \langle n_1, n_4\rangle, \langle n_2, n_5\rangle\}$. Note that although n_2 occurs both in the first set of $\vec{X}$ and the second set, we only we only process it once.

- The contexts that we compute for each pair in S is as follows:

$$\begin{array}{rcl} \langle n_1, n_3 \rangle & \rightarrow & \langle n_3, 1, 2 \rangle \\ \langle n_1, n_4 \rangle & \rightarrow & \langle n_4, 2, 2 \rangle \\ \langle n_2, n_5 \rangle & \rightarrow & \langle n_5, 1, 1 \rangle \end{array}$$

- So, when we call EVAL-BOOLEXP in line 12 to evaluate the predicate $\mathsf{position}() = 2$, $\vec{c} = (\langle n_3, 1, 2\rangle, \langle n_4, 2, 2\rangle, \langle n_5, 1, 1\rangle)$. On this input, EVAL-BOOLEXP(defined in Section 4.2) will return the vector $(\mathsf{false}, \mathsf{true}, \mathsf{false})$. This indicates that when we evaluate the entire step from node n_1 the candidate node n_3 should not be returned, while candidate n_4 should be returned. Likewise, when we evaluate the entire step from node n_2 the candidate node n_5 should not be returned.

- S is then is updated to $\langle n_1, n_4\rangle$ in line 13. Hence, the result computed in line 16 and returned in line 18 is $(\{n_4\}, \emptyset)$.

Exercise 1. Assume that *step* is the location step $\mathsf{ancestor} :: * \, [\mathsf{position}() = 3]$ and that t is the tree from Example 4. Describe the evaluation of PROCESS-STEP on $(\mathit{step}, t, \vec{X})$ where $\vec{X} = (\{n_6, n_5\}, \{n_7\})$. Phrase your answer in the same way as was done in Example 4. Does PROCESS-STEP generate contexts for EVAL-BOOLEXP with the same context node, but different context position and/or size?

4.2 Evaluating the other expressions

EVAL-NUMEXP, EVAL-STREXP, and EVAL-BOOLEXPshown in Algorithms 4, 5 and 6 all operate by a analysing the form of its input numeric expression and performing the corresponding action for each context in their input. They make calls to each other when necessary. The pseudocode should be self-explanatory.

4.3 Complexity

To analyze the total running time of evaluating a XPath expression, we introduce the following definition.

Definition 4. Define *size*(e) to be the total length of XPath expression e, when written down as a string.

Example 5. To illustrate, the *seqexp*expression $e =$ child::a[position()=1]/descendant::b has *size*$(e) = 37$.

Note that we always have $\#t \leq \mathit{size}(t)$ and $\#e \leq \mathit{size}(e)$ (when e is a location path). The following theorem shows that EVAL-SEQEXP runs in polynomial time in both *size*(e) and *size*(t).

Theorem 2. *Let e be a sequence expression, let t be an XML tree, and let $\vec{c} = (\langle x, p, s\rangle)$ consist of a single context. Then* EVAL-SEQEXP$(e, t, \vec{c})$ *runs in time* $\mathcal{O}(\mathit{size}(t)^4 \cdot \mathit{size}(e)^2)$.

Readers interested in the proof of this theorem are referred to [3]. (It is not required for passing the exam.) It should be stressed again that the exponential time behavior of NAIVE is avoided by evaluating the steps and predicates for the necessary contexts *together* instead of one-by-one.

Algorithm 4: EVAL-NUMEXP$(e, t, \vec{X})$

```
    Input: a numexp $e$, XML tree $t$, and a variable-length of contexts $\vec{c}$;
    Result: a vector of numbers $(n_1, \dots, n_k)$ of the same length as $\vec{c}$ such that $n_i$ is the
            result of evaluating $e$ on the $i$-th context in $\vec{c}$ on $t$
 1  begin
 2      Let $\vec{c} = (\langle x_1, p_1, s_1\rangle, \langle x_2, p_2, s_2\rangle, \dots, \langle x_k, p_k, s_k\rangle)$
 3      if $e$ is position() then
 4          return $(p_1, \dots, p_k)$
 5      else if $e$ is last() then
 6          return $(s_1, \dots, s_k)$
 7      else if $e$ is count($e'$) then                                  /* $e'$ is a seqexp */
 8          Let $(X_1, \dots, X_k)$ := EVAL-SEQEXP$(e', t, \vec{c})$
 9          return $(|X_1|, \dots, |X_k|)$
10      else if $e$ is number() then
11          return $(to\_num(strval(t, x_1)), \dots, to\_num(strval(t, x_k)))$
12      else if $e$ is a numeric literal $n$ then
13          return $(n, \dots, n)$                                          /* $k$ times */
14      else if $e$ is $e_1$ op $e_2$ with op $\in \{+, -, *, div\}$ then
15          $(n_1, \dots, n_k)$ := EVAL-NUMEXP$(e_1, t, \vec{c})$
16          $(m_1, \dots, m_k)$ := EVAL-NUMEXP$(e_2, t, \vec{c})$
17          return $(n_1$ op $m_1, \dots, n_k$ op $m_k)$
18      end
19  end
```

Exercise 2. To see how EVAL-SEQEXP differs from *naive*, reconsider in particular the family of location paths $\{Q_1, Q_2, \dots\}$ defined in the proof of Proposition 1.

1. First transform Q_i to be a valid expression in our fragment.
2. Then describe (in global terms) the evaluation of EVAL-SEQEXP on $(Q_1, t, \vec{c})$ when t is the tree in Example 4 and $\vec{c}$ is the vector $(\langle n_0, 1, 1\rangle)$ of length one. In particular why don't we get the exponential-time behavior of NAIVE?

(This exercise will be corrected in the exercise session on XML Schemas.)

5 Evaluating axes and node tests

To complete the definition of PROCESS-STEP, it remains to show how to efficiently compute, given a set of nodes X, all nodes that are reachable from X by an axis α and that satisfy a test τ. (This is necessary for line 2 of PROCESS-STEP). THIS SECTION IS PROVIDED FOR ILLUSTRATION PURPOSES ONLY. IT IS NOT REQUIRED MATERIAL FOR THE EXAM

5.1 Evaluating axes

Remember from Section 2 that given a node x, we only have the operations *firstchild*(x, t), *nextsib*(x, t), *parentnode*(x, t), and *prevsib*(x, t) to compute the nodes that are immediate neighbors of x. We hence need a way to compute the axes child, descendant, etc.

Algorithm 5: EVAL-STREXP$(e, t, \vec{c})$

```
Input: a strexp $e$, XML tree $t$, and variable-length vector of contexts $\vec{c}$;
Result: a vector of strings $(s_1, \ldots, s_k)$ of the same length as $\vec{c}$ such that $s_i$ is the
        result of evaluating $e$ on the $i$-th context in $\vec{c}$ on $t$
begin
    Let $\vec{c} = (\langle x_1, p_1, s_1\rangle, \langle x_2, p_2, s_2\rangle, \ldots, \langle x_k, p_k, s_k\rangle)$
    if $e$ is string() then
        return $(strval(t, x_1), \ldots, strval(t, x_2))$;
    else if $e$ is name() then
        return $(name(t, x_1), \ldots, name(t, x_2))$
    else if $e$ is string($e'$) then                          /* $e'$ is a seqexp */
        Let $(X_1, \ldots, X_k) :=$ EVAL-SEQEXP$(e', t, \vec{c})$
        for $i := 1$ to $k$ do
            if $X_i$ is a singleton set $\{y_i\}$ then
                $s_i := strval(t, y_i)$
            else raise error
        end
        return $(s_1, \ldots, s_k)$
    else if $e$ is name($e'$) then                            /* $e'$ is a seqexp */
        Let $(X_1, \ldots, X_k) :=$ EVAL-SEQEXP$(e', t, \vec{c})$
        for $i := 1$ to $k$ do
            if $X_i$ is a singleton set $\{y_i\}$ then
                $s_i := name(t, y_i)$
            else raise error
        end
        return $(s_1, \ldots, s_k)$
    else if $e$ is a string literal $s$ then
        return $(s, \ldots, s)$                                    /* $k$ times */
    end
end
```

Without loss of generality, we extend the operations *firstchild*, *nextsib*, *parentnode*, and *prevsib* to work on sets of nodes:

$$\begin{aligned} \mathit{firstchild}(X, t) &:= \{\mathit{firstchild}(x, t) \mid x \in X\} \\ \mathit{parentnode}(X, t) &:= \{\mathit{parentnode}(x, t) \mid x \in X\} \\ \mathit{nextsib}(X, t) &:= \{\mathit{nextsib}(x, t) \mid x \in X\} \\ \mathit{prevsib}(X, t) &:= \{\mathit{prevsib}(x, t) \mid x \in X\} \end{aligned}$$

Since *firstchild*(x, t), *parentnode*(x, t), *nextsib*(x, t), and *prevsib*(x, t) all run in constant time, we can compute *firstchild*(X, t), *nextsib*(X, t), *parentnode*(X, t), and *prevsib*(X, t) in linear time $\mathcal{O}(|X|)$ simply by iterating over X and calling the corresponding node operation.

PROCESS-AXIS, shown in Algorithm 7, then takes as input an axis α, a tree t, and a set of nodes X, and computes the set $\{y \mid y$ reachable from some $x \in X$ by following the axis $\alpha\}$. It uses the auxiliary function CLOSE(Ops, t, X) where $Ops \subseteq \{\mathit{firstchild}, \mathit{nextsib}, \mathit{parentnode},$

Algorithm 6: EVAL-BOOLEXP$(e, t, \vec{c})$

```
Input: a boolexp $e$, XML tree $t$, and variable-length vector of contexts $\vec{c}$;
Result: a vector of booleans $(b_1, \ldots, b_k)$ of the same length as $\vec{c}$ such that $b_i$ is the
        result of evaluating $e$ on the $i$-th context in $\vec{c}$ on $t$
begin
    Let $\vec{c} = (\langle x_1, p_1, s_1\rangle, \langle x_2, p_2, s_2\rangle, \ldots, \langle x_k, p_k, s_k\rangle)$
    if $e$ is true() then
        return (true, ..., true)
    else if $e$ is false() then
        return (false, ..., false)
    else if $e$ is $e_1$ op $e_2$ with op $\in$ {and, or} then
        Let $(b_1, \ldots, b_k)$ := EVAL-BOOLEXP$(e_1, t, \vec{c})$
        Let $(b'_1, \ldots, b'_k)$ := EVAL-BOOLEXP$(e_2, t, \vec{c})$
        return $(b_1$ op $b'_1, \ldots, b_k$ op $b'_k)$
    else if $e$ is not$(e_1)$ then
        Let $(b_1, \ldots, b_k)$ := EVAL-BOOLEXP$(e_1, t, \vec{c})$
        return $(not(b_1), \ldots, not(b_k))$
    else if $e$ is fn:empty$(e')$ then                          /* $e'$ is a seqexp */
        Let $(X_1, \ldots, X_k)$ := EVAL-SEQEXP$(e', t, \vec{c})$
        return $(empty?(X_1), \ldots, empty?(X_n))$
    else if $e$ is $e_1 = e_2$ then
        if $e_1$ and $e_2$ are both strexp then
            Let $(s_1, \ldots, s_k)$ := EVAL-STREXP$(e_1, t, \vec{c})$
            Let $(s'_1, \ldots, s'_k)$ := EVAL-STREXP$(e_2, t, \vec{c})$
            return $(streq(s_1, s'_1), \ldots, streq(s_k, s'_k))$
        else                                         /* $e_1$ and $e_2$ are both numexp */
            Let $(n_1, \ldots, n_k)$ := EVAL-NUMEXP$(e_1, t, \vec{c})$
            Let $(n'_1, \ldots, n'_k)$ := EVAL-NUMEXP$(e_2, t, \vec{c})$
            return $(numeq(n_1, n'_1), \ldots, numeq(n_k, n'_k))$
    end
end
```

prevsib} is a set of operations, as shown in Algorithm 8. CLOSE computes the *closure of X by the operations in Ops*, which is the smallest set Y such that:

1. $X \subseteq Y$;
2. for each $y \in Y$ and each $op \in Ops$, also $op(t, y) \in Y$

For instance, the closure of the set $\{x\}$ by $\{parentnode\}$ yields the set of all ancestors of $\{x\}$. Moreover, the closure of $\{x\}$ by $\{firstchild, nextsib\}$ yields the set of all descendants of x.

Proposition 3. CLOSE(Ops, t, X) *can be evaluated in $\mathcal{O}(\#t)$ time.*

Proof. Represent the set Z of nodes that we have already processed as an array of bits, one bit for each node in t; the bits set to true represent the nodes in Z, the bits set to false represent the nodes not in Z. Membership in Z can then be decided in constant time (we assume that

Algorithm 7: PROCESS-AXIS$(\alpha, t, \vec{c})$

```
Input: an axis $\alpha$; a tree $t$; and a nodeset $X$;
Result: the set of nodes $\{y \mid y$ reachable from some $x \in X$ by following the axis $\alpha\}$
begin
    Initialize result $Y := \emptyset$
    if $\alpha$ is self then set $Y := X$
    else if $\alpha$ is child or attribute then set $Y :=$ CLOSE$(\{nextsib\}, t, firstchild(X, t)))$
    else if $\alpha$ is parent then set $Y := parentnode(X, t))$
    else if $\alpha$ is descendant then set $Y :=$ CLOSE$(\{firstchild, nextsib\}, t, firstchild(X, t))$
    else if $\alpha$ is ancestor then set $Y :=$ CLOSE$(parentnode, t, parentnode(X, t))$
    else if $\alpha$ is descendant-or-self then set $Y := X \cup$ PROCESS-AXIS(descendant$, t, X)$
    else if $\alpha$ is ancestor-or-self then set $Y := X \cup$ PROCESS-AXIS(ancestor$, t, X)$
    else if $\alpha$ is following then
        set $Y :=$ EVAL-AXIS(ancestor-or-self$, t, X)$
        set $Y := nextsib(Y, t)$
        set $Y :=$ CLOSE$(nextsib, t, Y)$
        set $Y :=$ EVAL-AXIS(descendant-or-self$, t, Y)$
    else if $\alpha$ is preceding then
        set $Y :=$ EVAL-AXIS(ancestor-or-self$, t, X)$
        set $Y := prevsib(Y, t)$
        set $Y :=$ CLOSE$(prevsib, t, Y)$
        set $Y :=$ EVAL-AXIS(descendant-or-self$, t, Y)$
    else if $\alpha$ is following-sibling then set $Y :=$ CLOSE$(nextsib, t, nextsib(Y, t))$
    else                                           /* $\alpha$ is preceding-sibling */
        set $Y :=$ CLOSE$(prevsib, t, prevsib(Y, t))$
    /* Now filter from $Y$ the nodes of the correct kind                  */
    if $\alpha$ is attribute then
        remove from $Y$ all nodes that are text or element nodes
    else
        remove from $Y$ all nodes that are attribute nodes
    end
end
```

array accesses take constant time). Now observe that, since we take care to process each node at most once, the while loop in line 4 is executed at most as many times as there are nodes in t (i.e., at most $\#t$ times). Each invocation of the body of the while loop requires us to

- pop the queue, which takes constant time;
- set the bit flag for y in Z to true, which takes constant time;
- execute all operators in *Ops* on the current node y. There are at most 4 such operators (namely *firstchild*, *nextsib*, *parentnode*, *prevsib*), each of which can be executed in constant time according to Section 2. Hence, this also takes constant time;
- check whether $op(t, y)$ is already in Z. As explained above, this takes constant time;

Algorithm 8: CLOSE(Ops, t, X)

```
   Input: a set of operations $Ops \subseteq \{firstchild, nextsib, parentnode, prevsib\}$; a tree $t$; and
          a nodeset $X$;
   Result: the closure of $X$ by $Ops$
1  begin
2      Initialize result set $Y := X$
3      Initialize set of prossessed nodes $Z := \emptyset$
4      Initialize a queue $Q$ and add all $X$ to $Q$
       /* $Q$ contains all nodes that we have not processed yet            */
5      while $Q$ is not empty do
6          Pop a node $y$ from $Q$
7          Add $y$ to $Z$
8          foreach $op$ in $Ops$ do
9              if $op(t, y) \notin Z$ then
10                 Add $op(t, y)$ to $Y$
11                 Push $op(t, y)$ on $Q$
12             end
13         end
14     end
15     return $Y$
16 end
```

- possibly add $op(t, y)$ to Q and Y, which takes constant time if (1) we represent Q as a linked list and (2) we also represent Y as an array of bits.

In conclusion, lines 6-13 take constant time and are executed at most $\#t$ times. Hence, CLOSE runs in time $\mathcal{O}(\#t)$. □

Proposition 4. PROCESS-AXIS($axis, t, X$) *runs in time* $\mathcal{O}(\#t)$. *In other words:* PROCESS-AXIS *runs in linear time.*

Proof. A careful inspection of PROCESS-AXIS reveals that for each *axis*we only make a *constant* number of calls of the form CLOSE(Ops, t, Z) and $op(Z, t)$. (Indeed, although some axes recursively call PROCESS-AXISwith another axis, this recursion always ends immediately.) Each CLOSE(Ops, t, Z) runs in time $\mathcal{O}(\#t)$ by Proposition 3. Moreover, we have already observed that each $op(Z, t)$ runs in time $\mathcal{O}(|Z|)$, and hence, since Z is always a subset of the nodes in t (i.e., $|Z| \leq \#t$), also in time $\mathcal{O}(\#t)$. In conclusion, we execute a constant number of calls, each which runs in $\mathcal{O}(\#t)$ time. Lines 23-27 can be executed by iterating over all elements in Y and dropping those elements that are not of the correct type. (Recall from Section 2 that we can determine the type of an element in constant time.) Since Y is a subset of the nodes in t, this can hence be done in $\mathcal{O}(\#t)$ steps. As such, the total evaluation runs in $\mathcal{O}(\#t)$ time, as claimed. □

5.2 Evaluating tests

Once we have computed the set of all nodes Y reachable from the set of nodes X by axis α, we can iterate over Y and using the operations *name*, *is_elemnode*, etc. from Section 2 to retain only those nodes that satisfy a given node test τ. Since Y is a subset of the nodes in t (hence $|Y| \leq \#t$) and since *name*, *is_elemnode* etc all take time at most $\mathcal{O}(size(t))$, the total time needed to compute the nodes in Y that satisfy τ is $\mathcal{O}(size(t)^2)$. Combined with Proposition 4 we obtain:

Corollary 5. *The set S in line 2 of Algorithm* PROCESS-STEP*can be computed in time $\mathcal{O}(size(t)^2)$*

References

[1] Anders Berglund, Scott Boag, Don Chamberlin Mary F. Fernndez, Michael Kay, Jonathan Robie, and Jrme Simon. Xml path language (xpath) 2.0. Technical report, World Wide Web Consortium, 2007. http://www.w3.org/TR/xpath20/.

[2] James Clark and Steve DeRose. Xml path language (xpath) version 1.0. Technical report, World Wide Web Consortium, 1999. http://www.w3.org/TR/xpath/.

[3] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient algorithms for processing xpath queries. *ACM Transactions on Database Systems*, 30(2):444–491, 2005.