http://webdam.inria.fr/

Web Data Management

XPath and XQuery

Serge Abiteboul
INRIA Saclay & ENS Cachan

Ioana Manolescu
INRIA Saclay & Paris-Sud University

Philippe Rigaux
CNAM Paris & INRIA Saclay

Marie-Christine Rousset
Grenoble University

Pierre Senellart
Télécom ParisTech

Contents

1 Introduction

This chapter introduces XPath and XQuery, two related languages that respectively serve to navigate and query XML documents. XPath is actually a subset of XQuery. Both languages, specified by the W3C, are tightly associated, and share in particular the same conceptual modeling of XML documents. Note that the XPath fragment of XQuery has a well-identified purpose (expressing "paths" in an XML tree) and as such can be used independently in other XML processing contexts, such as inside the XSLT transformation language. XQuery uses XPath as a core language for path expressions and navigation.

XQuery is a declarative language, and intends to play for XML data the role of SQL in the relational realm. At a syntactical level, it is somewhat inspired from SQL. More importantly, it is expected to benefit from a mixture of physical storage, indexing, and optimization techniques, in order to retrieve its result by accessing only a small fraction of its input. XQuery constitutes therefore an appropriate choice when large XML documents or large collections of documents must be manipulated.

In this chapter, we use as running example a *movies* XML database. Each XML document represents one movie, and is similar in structure to the sample document shown in Figure 1.

We begin the chapter with a bird's eye view of XQuery principles, introducing the XML data model that supports the interpretation of path expressions and queries, and showing the main features of the two languages. We then consider in more detail XPath and XQuery in a rather informal way. Finally we reconsider XPath more formally, investigating nice connections with first-order logic.

2 Basics

The W3C devoted a great deal of effort (along with heavy documents) to formally define the data model that underlies the interpretation of XPath and XQuery expressions. We just need, for the purpose of this introduction, to understand that XQuery is designed as the database query language for XML sources. As such, it must fulfill some basic requirements, two of the most important being that:

1. there exists a well-defined "data model", i.e., a set of constructs and typing rules that dictate the shape of any information that conceptually constitutes an XML database;
2. the query language is closed (or *composable*): in plain English, this means that queries operate on instances of the data model, and produce instances of the data model.

Let us first consider the corresponding requirements for relational databases. In a relational database, data are represented using two-dimensional "tables". Each table consists of a set of rows with a predefined list of "columns". Given a row and a column, an entry consists of an atomic value of a predefined type specified by the column. This constitutes a simple and effective data model. Regarding the SQL language, each query takes one or several tables as input and produces one table as output. (We ignore here some features such as ordering the rows with **order by** commands.) Even if the query returns a single value, this value is seen as a cell in a one-row, one-column, result table. The closed-form requirement guarantees that queries can be composed to form complex expressions. In other words, one can build complex queries using composition because the output of a query can serve as input to another one.

Let us now consider these requirements in the context of XML. We must be able to model the content of the documents, which is much more flexible and complex than the content of a relational table. We must also model the structure of the database as a set of documents, with possibly quite different contents and structures. And, finally, we need to make sure that any query output is also a collection of XML documents, so that we can compose queries.

A difficulty is that we sometimes want to talk about a tree and we sometimes want to focus on a sequence of trees (the children of a node in a tree). The W3C has therefore introduced a data model which, beyond the usual atomic data types, proposes two constructs: *trees* to

```
<?xml version="1.0" encoding="UTF-8"?>

<movie>
  <title>Spider-Man</title>
  <year>2002</year>
  <country>USA</country>
  <genre>Action</genre>
  <summary>On a school field trip, Peter Parker (Maguire) is
    bitten by a genetically modified spider. He wakes
    up the next morning with incredible powers. After
    witnessing the death of his uncle (Robertson),
    Parkers decides to put his new skills to use in
    order to rid the city of evil, but someone else
    has other plans. The Green Goblin (Dafoe) sees
    Spider-Man as a threat and must dispose of him. </summary>
  <director id='21'>
    <last_name>Raimi</last_name>
    <first_name>Sam</first_name>
    <birth_date>1959</birth_date>
  </director>
  <actor id='19'>
    <first_name>Kirsten</first_name>
    <last_name>Dunst</last_name>
    <birth_date>1982</birth_date>
    <role>Mary Jane Watson</role>
  </actor>
  <actor id='22'>
    <first_name>Tobey</first_name>
    <last_name>Maguire</last_name>
    <birth_date>1975</birth_date>
    <role>Spider-Man / Peter Parker</role>
  </actor>
  <actor id='23'>
    <first_name>Willem</first_name>
    <last_name>Dafoe</last_name>
    <birth_date>1955</birth_date>
    <role>Green Goblin / Norman Osborn</role>
  </actor>
</movie>
```

Figure 1: An XML document describing a movie

model the content of XML documents, and *sequences* to represent any ordered collection of "items", an item being either an atomic value or a document.

Another difficulty is that, as we shall see, we sometimes want to talk about collections without duplicates. For instance, the result of the simplest XPath queries is such a collection. Indeed, the specification of XPath 1.0, which is still the most widely implemented version of the language, does not allow arbitrary sequences, but only *node sets*, duplicate-free collections of nodes. So we shall have to carefully distinguish between sequences (ordered lists possibly with duplicates) and duplicate-free collections or node sets.

To conclude this preliminary discussion, we want to stress that XQuery is a functional language based on *expressions*: any expression takes sequences as inputs and produces a sequence as output. This is probably everything that needs to be remembered at this point. We now illustrate the principles, starting with the tree model of XML documents.

2.1 XPath and XQuery data model for documents

In the XQuery model, an XML document is viewed as a tree of *nodes*. Each node in a tree has a *kind*, and possibly a *name*, a *value*, or both. These concepts are important for the correct interpretation of path expressions. Note that this is actually a simplified version of the object-based representation that supports the **Dom** API (see Chapter **??**). Here is the list of the important node kinds that can be found in an XML tree:

- **Document**: the *root node* of the XML document, denoted by "/";
- **Element**: element nodes that correspond to the tagged nodes in the document;
- **Attribute**: attribute nodes attached to **Element** nodes;
- **Text**: text nodes, i.e., untagged leaves of the XML tree.

The data model also features **ProcessingInstruction**, **Comment**, and **Namespace** node kinds. The first two can be addressed similarly as other nodes, and the third one is used for technical processing of namespaces that is rarely needed. Therefore, to simplify, we do not consider these node kinds in the following presentation. Another important feature of the XQuery data model is the *data type* that can be attached to element and attribute nodes. This data type comes from an XML Schema annotation (see Chapter **??**) of the document. It is a very powerful feature that allows XQuery queries to deal differently with nodes of different declared data types. It also allows for the static verification of a query. However, because of lack of support from implementations, this component of XQuery is sparingly used. Again to simplify, we mostly ignore data types in the remaining of this chapter.

It is worth mentioning that the tree model ignores syntactic features that are only relevant to serialized representations. For instance, literal sections or entities do not appear, since they pertain to the physical representation and thus have no impact on the conceptual view of a document. Entities are supposed to have been resolved (i.e., references replaced by the entity content) when the document is instantiated from its physical representation.

Figure 2 shows a serialized representation of an XML document, and Figure 3 its interpretation as an XML tree. The translation is straightforward, and must be understood by anyone aiming at using XPath or XQuery. Among the few traps, note that the typical fragment `<a>v</a>` is *not* interpreted as a single node with name `a` and value `v`, but as two nodes: an

```
<?xml version="1.0"
      encoding="utf-8"?>
<A>
  <B att1='1'>
    <D>Text 1</D>
    <D>Text 2</D>
  </B>
  <B att1='2'>
    <D>Text 3</D>
  </B>
  <C att2="a"
     att3="b"/>
</A>
```

Figure 2: Example XML document in serialized form

Element which bears the name, and a **Text** child which bears the value. It is important to keep in mind a few other characteristics which are common to all tree representations, and help understand the meaning of expressions:

- the *document order* denotes the order of the nodes when the tree is traversed in pre-order; it is also the order of the serialized representation;
- a tree has a unique **Document** node, called the *root node* of the tree in the following; this root node has a unique child of type **Element**, called the *root element*.

A root node may also have other children such as comments or processing instructions but as previously mentioned, we ignore them here. Next, for each node in a tree, the concepts of *name* and *value* are defined as follows: (i) an **Element** node has a name (i.e., the tag in the serialized representation), but no value[1]; (ii) a **Text** node has a value (a character string), but no name; and (iii) an **Attribute** node has both a name and a value. As we shall see **Attribute** nodes are special: attributes are not considered as first-class nodes in an XML tree and are addressed in a specific manner.

A term commonly used is “content” which must be distinguished from the notion of “value”. Although an **Element** node N has no value, it has a *content*, which is the XML subtree rooted at N. If we consider the serialized representation instead, the content is (equivalently) the part of the document contained between the opening and closing tags of the element. Now one often makes the mistake to see the content of an XML node as the serialized representation. It is important to keep in mind that conceptually it is a tree. To increase the confusion, one sometimes speak of the *textual content* of a node N, which is the concatenation of the values of the **Text** nodes which are descendant of N. In others words, the textual content of N is obtained from its content by getting rid of all the structural information. This makes sense only when we think of an XML document as structured text.

Although all this may seem confusing at first glance, it is important to be very comfortable with these notions and in particular keep in mind that *the content of a node of an XML tree is the*

[1]No value *per se*; the XPath recommendation defines the value of an element node as the concatenation of the values of all **Text** nodes below

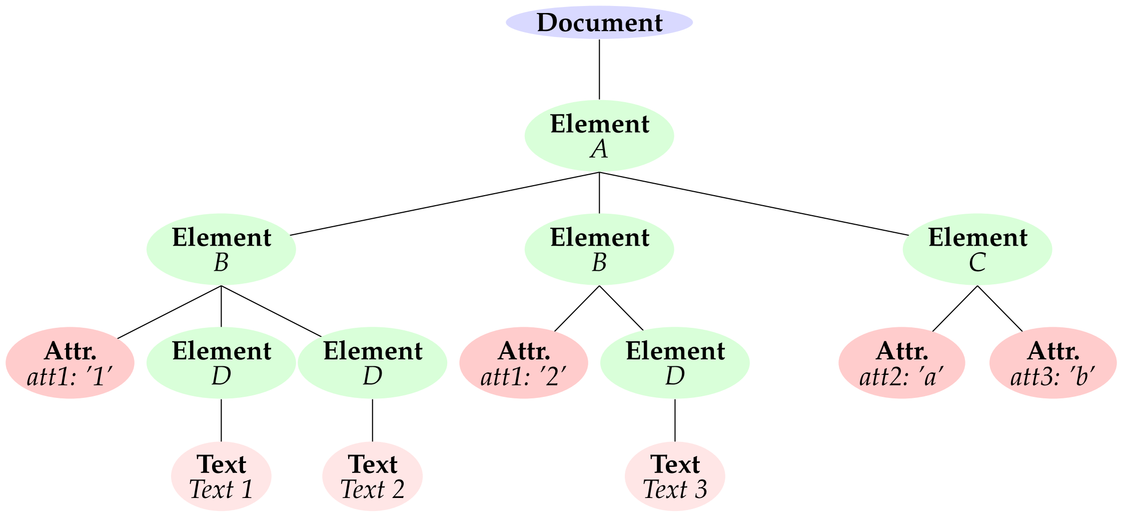

Figure 3: Tree representation of the XML document from Figure 2

subtree rooted at that node.

2.2 The XQuery model (continued) and sequences

The main construct manipulated by XQuery expressions is the *sequence of items*, a deliberately vague and general structure that covers all kinds of information that can be dealt with in an XML database. An item is either an *atomic value* or a *node*. In the latter case, when the node N is an **Element** or a **Document** (i.e., the root node of a document), it represents the whole XML tree rooted at N.

Sequences constitute a central concept for XQuery, since a query takes as input one or more sequences and produces as output a sequence.

A sequence may be an enumeration, surrounded with parentheses. The content of a sequence may also be described intentionally (e.g., all integers between 1 and 5.)

```
(1, 'a', 1, 'zgfhgf', 2.12)
(1 to 5)
```

Observe that the first sequence mixes integers, characters, character strings, floating numbers. The mixture may also contain nodes and accepts duplicates. Due to the very versatile shape of semi-structured information, the data model actually puts almost no restriction on the content of a sequence. An important point is that sequences cannot be embedded inside each other: a sequence is always a flat, ordered, collection of atomic values or nodes. In other words, the following two sequences are identical:

```
(1, (2, 3), (4, 5, 6))
(1, 2, 3, 4, 5, 6)
```

Since querying atomic values is of little interest, a query takes in general as input XML documents or a collection of XML documents. A *collection* is actually nothing else than a persistent sequence of XML documents which can be referred to by a name. XQuery identifies its input(s) with the following functions:

1. *doc*() takes the URI of an XML document and returns a singleton document tree;
2. *collection*() takes the URI of a collection of XML documents and returns a sequence of trees.

For instance,

```
doc('Spider-Man.xml')
collection('movies')
```

The result of *doc('Spider-Man.xml')* is the singleton sequence consisting of the *root node* of the tree representation of the XML content found in `Spider-Man.xml`. The node kind is **Document**.

As part of our running example, the *movies* collection contains a set of XML documents, each describing a specific movie. The result of *collection('movies')* is the sequence of root nodes of the collection of *movie* documents. In general, the *collection*() function returns a sequence of items. Although its organization is much more flexible, a collection is somehow comparable to tables in the relational model, where items of the collection set play the role of tuples.

The functions *doc*() and *collection*() take as input a URI. They can therefore be used to access a database that is stored either locally or remotely. For instance, the URI *movies* may refer to the database serving all the movie XML documents. In both cases, the output is a sequence of **Document** nodes. Given such sequences available through calls to the *doc*() or *collection*() functions, XPath and XQuery expressions can be expressed to retrieve information from these contents. Such an environment is typically an XML database system, e.g., the EXIST system (see Chapter **??**).

2.3 Specifying paths in a tree: XPath

XPath is a syntactic fragment of XQuery, which forms the basic means of navigating in an XML tree. At its core are *path expressions* that denote paths in a tree, using a mixture of structural information (node names, node kinds) and constraints on data values. Here is a first example:

```
doc('Spider-Man.xml')/movie/title
```

An XPath expression consists of *steps*, separated by "/". The above expression consists of three steps. The first one returns a singleton with the root node of the document. The second step (`movie`) returns the children of the root node with name `movie`. Again, this is a singleton since the root node only has one **Element** child. Finally, the third step (`title`) returns the sequence of **Element** nodes, of name `title`, children of the `movie` element. The sequence of `title` nodes is the result of the whole XPath expression.

More generally, a path expression is evaluated with respect to a *context node*, which is often (but not always) the root node of some XML document, and its result is a sequence of terminal nodes of the paths that start from the context node and match the expression.

So far, the interpretation is quite similar to the usual navigation in the directory tree of a computer system. XPath is more expressive and permits very flexible navigation in the trees with access to both content and structure of the visited trees. The following example features a *predicate*, i.e., a Boolean expression that must be satisfied for the nodes to be qualified in the result sequence. The interpretation should be clear: one retrieves the nodes corresponding to the actresses of the input document whose last name is `Dunst`.

```
doc('Spider-Man.xml')/movie/actor[last_name='Dunst']
```

One obtains a sequence, with as many `actor` items as there are matching nodes in the document (here: only one). Note that the item is an **Element** node, *along with its content*, i.e., the subtree at this node. In other word, the (serialized) result is:

```
<actor id='19'>
  <first_name>Kirsten</first_name>
  <last_name>Dunst</last_name>
  <birth_date>1982</birth_date>
  <role>Mary Jane Watson</role>
</actor>
```

The comparison with navigation in file system directories can be extended a little further. Indeed, XPath is not limited to going down the tree, following the "child" axis, but can also access the (unique) parent of a node. The following XPath expression gives all the `title`s of movies in the `movies` collection, featuring Kirsten Dunst as an actress.

```
collection('movies')/movie/actor[last_name='Dunst']/../title
```

To better understand what is going on here, it is probably useful to take a representation of the tree of a specific movie (say, Spider-Man), and draw the path that matches the above expression (knowing that, as expected, the "`..`" step denotes the parent of the context node). There exists an equivalent (and maybe more natural) expression:

```
collection('movies')/movie[actor/last_name='Dunst']/title
```

The power of XPath is however relatively limited in term of node retrieval.[2] Moreover, the result of an XPath expression can only consist of a sequence of nodes from the input document. This is a very severe restriction since it prevents the construction of new XML documents. However it constitutes a convenient tool for describing classes of paths in XML trees, and can be used together with more powerful languages to obtain complex queries.

[2]This is all the truer if one restricts the language to the XPath 1.0 fragment, that cannot express much more than these kinds of path expressions. XPath 2.0, with its iteration features described further, is more powerful, but still limited compared to XQuery.

2.4 A first glance at XQuery expressions

XQuery is a functional language. An expression is a syntactic construct which operates on a sequence (the input) and produces a sequence (the output). Since the output of an expression can be used as the input of another expression, the combination of expressions yields the mechanism to create very complex queries.

The simplest expression is a literal: given a sequence S, it returns S. The following is therefore a valid XQuery expression:

```
(1, 'a', 1, 'zgfhgf', 2.12)
```

XQuery becomes more powerful than XPath when it comes to constructing rich output or to expressing complex statements. The following simple examples illustrate the most important features without delving into details.

First, XQuery allows the construction of new documents, whose content may freely mix literal tags, literal values, and results of XQuery expressions. The following shows the construction of an XML document containing the list of movie titles.

```
document {
  <titles>
    {collection('movies')//title}
  </titles>
}
```

The *collection*() function is now embedded in an XML literal fragment (formed here of a single root element `titles`). Expressions can be used at any level of a query, but in order to let the XQuery parser recognize an expression *e* which must be evaluated and replaced by its result, the expression *e* must be surrounded by curly braces {} when it appears inside literal elements. Forgetting the braces results in a literal copy of the expression in the result (i.e., it remains uninterpreted). Any number of expressions can be included in a template, thereby giving all freedom to create new XML content from an arbitrarily large number of XML inputs.

Note that, in the above query, XPath is used as a core language to denote paths in an existing XML document referred to by the *doc*() expression.

Here is a second example of a powerful XQuery expression that goes far beyond the capabilities of simple path expressions. The following shows a query that returns a list of character string with the title of a movie (published after 2005) and the name of its director.

```
for $m in collection('movies')/movie
where $m/year >= 2005
return
<film>
  {$m/title/text()},
  director: {$m/director/last_name/text()}
</film>
```

The query is syntactically close to the SQL **select-from-where** construct. The **for** clause is similar to the SQL **from**, and defines the range of a variable `$m`. The **return** clause (in the spirit of SQL **select**) constructs the result, using variable `$m` as the root of XPath expression. The output obtained from our sample collection, is (disregarding whitespace):

```
<film>A History of Violence, director: Cronenberg</film>
<film>Match Point, director: Allen</film>
<film>Marie Antoinette, director: Coppola</film>
```

Note that the result is a sequence of nodes, and not an XML document.

Expressions based on the **for** clause are called *FLWOR expressions*. This is pronounced "flower" with the "F" standing for **for**, "L" for **let** (a clause not used in the previous example), "W" for **where**, "O"for **order by** (an optional ordering clause), and "R" for **return**. A FLWOR expression must contain at least one (but potentially many) **for** or **let** clause and exactly one **return** clause, the other parts being optional. The expressive power of the language comes from its ability to define variables in flexible ways (**from** and **let**), from supporting complex filtering (**where**) and ordering (**order by**), and allowing the construction complex results (**return**).

2.5 XQuery vs XSLT

XQuery is thus a choice language for querying XML documents and producing structured output. As such, it plays a similar role as XSLT, another W3C standardized language for transforming XML documents, that is presented in more detail in the companion Web site of this book. The role of XSLT is to extract information from an input XML document and to transform it into an output document, often in XML, which is also something that XQuery can do. Therefore, both languages seem to compete with one another, and their respective advantages and downsides with respect to a specific application context may not be obvious at first glance. Essentially:

- XSLT is good at *transforming* documents, and is for instance very well adapted to map the content of an XML document to an XHTML format in a Web application;
- XQuery is good at *efficiently retrieving* information from possibly very large repositories of XML documents.

Although the result of an XQuery query may be XML-structured, the creation of complex output is not its main focus. In a publishing environment where the published document may result from an arbitrarily complex extraction and transformation process, XSLT should be preferred.

Note however that, due to its ability to randomly access any part of the input tree, XSLT processors usually store in main memory the whole DOM representation of the input. This may severely impact the transformation performance for large documents. The procedural nature of XSLT makes it difficult to apply rewriting or optimization techniques that could, for example, determine the part of the document that must be loaded or devise an access plan that avoids a full main-memory storage. Such techniques are typical of *declarative* database languages such as XQuery that let a specialized module organize accesses to very large data sets in an efficient way.

We conclude here this introduction to the basics of XQuery. We next visit XPath in more depth.

3 XPath

The term *XPath* actually denotes two different languages for selecting nodes in a tree:

1. XPath 1.0, whose specification was finalized in 1999, is the most widely used version of XPath; implementations exist for a large variety of programming languages, and it is used as an embedded language inside another language in a number of contexts, especially in XSLT 1.0. XPath 1.0 is a simple language for navigating a tree, based on the notion of path expressions, and its expressive power is quite limited, as discussed further. Its data model is somewhat simpler than the XQuery data model discussed earlier in this chapter: node sets instead of sequences, and no data type annotations.

2. XPath 2.0, standardized in 2007, is an extension of XPath 1.0 that adds a number of commodity features, extends the data model to that of XQuery, and adds some expressiveness to the language, with the help of path intersection and complementation operators, as well as iteration features. XPath 2.0 is a proper subset of XQuery, and is also used inside XSLT 2.0. Apart from these two contexts, implementations of XPath 2.0 are rare. With a few technical exceptions, XPath 2.0 is designed to be backwards compatible with XPath 1.0: XPath 1.0 expressions are, mostly, valid XPath 2.0 expressions with the same results.

In this section, we mostly discuss XPath 1.0 and its core aspect, path expressions. We discuss briefly at the end of the section the additional features available in XPath 2.0. As already mentioned, a path expression consists of *steps*. It is evaluated over a list, taking each element of the list, one at a time. More precisely, a step is always evaluated in a specific *context*

$$[\langle N_1, N_2, \cdots, N_n \rangle, N_c]$$

consisting of a *context list* $\langle N_1, N_2, \cdots, N_n \rangle$ of nodes from the XML tree; and *a context node* N_c belonging to the context list, the node that is currently being processed. The result of a path expression, in XPath 1.0, is a *node set*. Here is a subtlety. The term *set* insists on the fact that there is no duplicate. Now to be able to be reused in another step, this set has to be turned into a sequence, i.e., be equipped with an order. We shall see how this is achieved.

3.1 Steps and path expressions

An XPath *step* is of the form:

$$\textit{axis}::\textit{node-test}\,[P_1]\;[P_2]\ldots[P_n]$$

Here, *axis* is an *axis name* indicating the direction of the navigation in the tree, *node-test* specifies a selection on the node, and each P_i ($n \geq 0$) is a *predicate* specifying an additional selection condition. A step is evaluated with respect to a *context*, and returns a *node set*. The following examples of steps illustrate these concepts:

1. `child::A` denotes all the **Element** children of the context node that have `A` for name; `child` is the axis, `A` is the node test (it restricts the selected elements based on their names) and there is no predicate. This very frequently used step can be denoted `A` for short.

2. `descendant::C[@att1=1]` denotes all the **Element** nodes descendant of the context node, named `C` and having an **Attribute** node `att1` with value 1. Observe how a node test is used to specify the name of the node and a predicate is used to specify the value of an attribute.

3. `parent::*[B]` denotes the parent of the context node, whatever its name may be (node test `*`) and checking it has an **Element** child named `B`. The predicate here checks the existence of a node. Since each node has a single parent, for a context node, the result is a collection of one element (the parent has a `B` child) or is empty (the test failed).

A path *expression* is of the form:

$$[/]step_1/step2/\ldots/step_n$$

When it begins with "`/`", it is an *absolute* path expression and the context of the first step is in that case the *root node*. Otherwise, it is a *relative* path expression. For a relative path expression, the context must be provided by the environment where the path evaluation takes place. This is the case for instance with XSLT where XPath expressions can be found in *templates*: the XSLT execution model ensures that the context is always known when a template is interpreted, and this context serves to the interpretation of all the XPath expressions found in the template.

The following are examples of XPath expressions:

1. `/A/B` is an *absolute* path expression which denotes the **Element** nodes with name `B`, children of the root element `A`;

2. `/A/B/@att1[.>2]` denotes all the **Attribute** nodes with name `att1` of the nodes obtained with the previous expression, whose values are greater than 2.

3. `./B/descendant::text()` is a *relative* path expression which denotes all the **Text** nodes descendant of an **Element** `B`, itself child of the context node.

In the last two expressions above, "`.`" is an abbreviation of the step `self::node()`, which refers to the context node itself. The axis `self` represents the "stay-here" navigation, and the `node()` node test is true for all nodes.

3.2 Evaluation of path expressions

The result of a path expression is a sequence of nodes obtained by evaluating successively the steps of the expression, from left to right. A step $step_i$ is evaluated with respect to the context of $step_{i-1}$. More precisely:

- For $i = 1$ (first step): if the path expression is *absolute*, the context is a singleton, the root of the XML tree; otherwise (for *relative* path expressions) the context is defined by the environment.

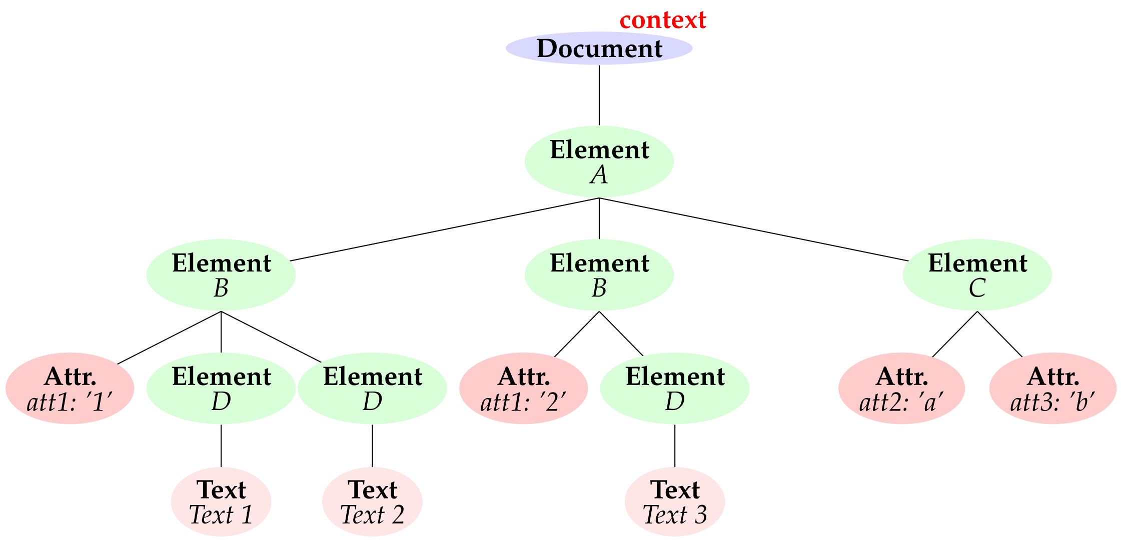

Figure 4: First step of the evaluation of `/A/B/@att1`

- For $i > 1$: if $\mathcal{N}_i = \langle N_1, N_2, \cdots, N_n \rangle$ is the result of step $step_{i-1}$, $step_i$ is successively evaluated with respect to the context $[\mathcal{N}_i, N_j]$, for each $j \in [1, n]$.

The result of the path expression is the node set obtained after evaluating the last step. As an example, consider the evaluation of `/A/B/@att1`. The path expression is absolute, so the context consists of the root node of the tree (Figure 4).

The first step, `A`, is evaluated with respect to this context, and results in the element node which becomes the context node for the second step (Figure 5).

Next, step `B` is evaluated, and the result consists of the two children of `A` named `B`. Each of these children is then taken in turn as a context node for evaluating the last step `@att1`.

1. Taking the first element `B` child of `A` as context node, one obtains its attribute `att1` (Figure 6);

2. Taking the second element `B` child of `A` as context node, one obtains its attribute `att1` (Figure 7).

The final result is the union of all the results of the last step, `@att1`. This is the union is a set-theoretic way, i.e., duplicates are eliminated. It is turned into a sequence (i.e., ordered) using an order that, as we shall see, is specified by the axis of the last step.

3.3 Generalities on axes and node tests

Given a context list, the axis determines a new context list. For each node in turn, the node test is evaluated filtering out some of the nodes. Then each predicate is evaluated and the nodes that fail some test are eliminated. This yields the resulting context list.

Table 1 gives the list of all XPath axes. Using them, it is possible to navigate in a tree, up, down, right, and left, one step or an arbitrary number of steps. As already mentioned, the axis also determines the order on the set of resulting nodes. It is in most cases the document

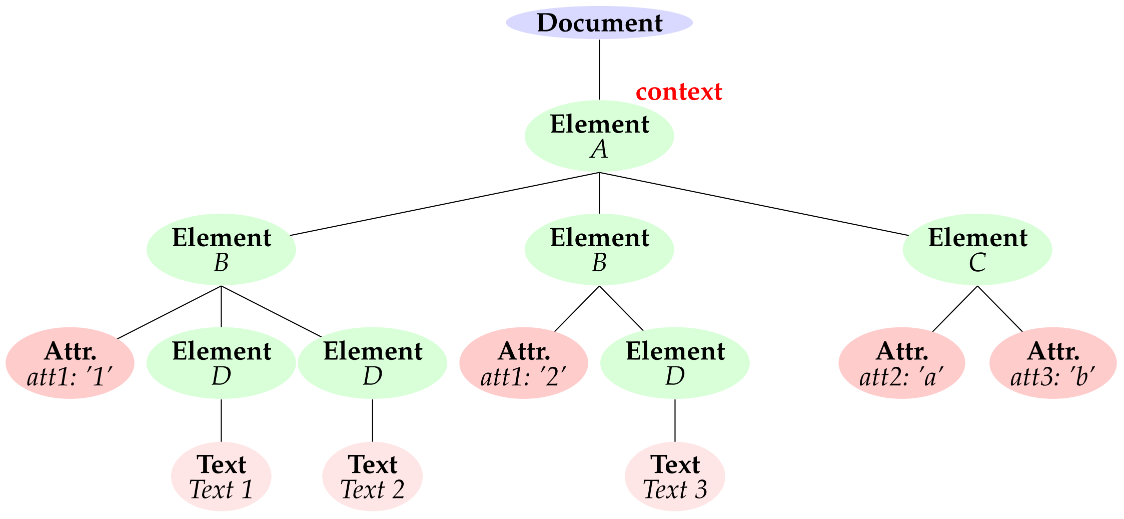

Figure 5: Second step of the evaluation of `/A/B/@att1`

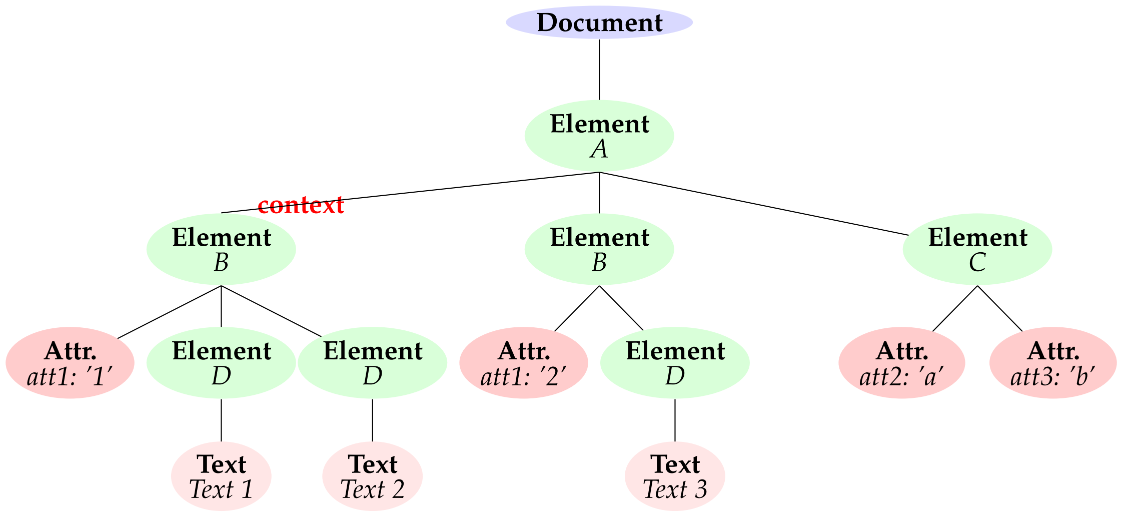

Figure 6: Evaluation of `@att1` with context node `B[1]`

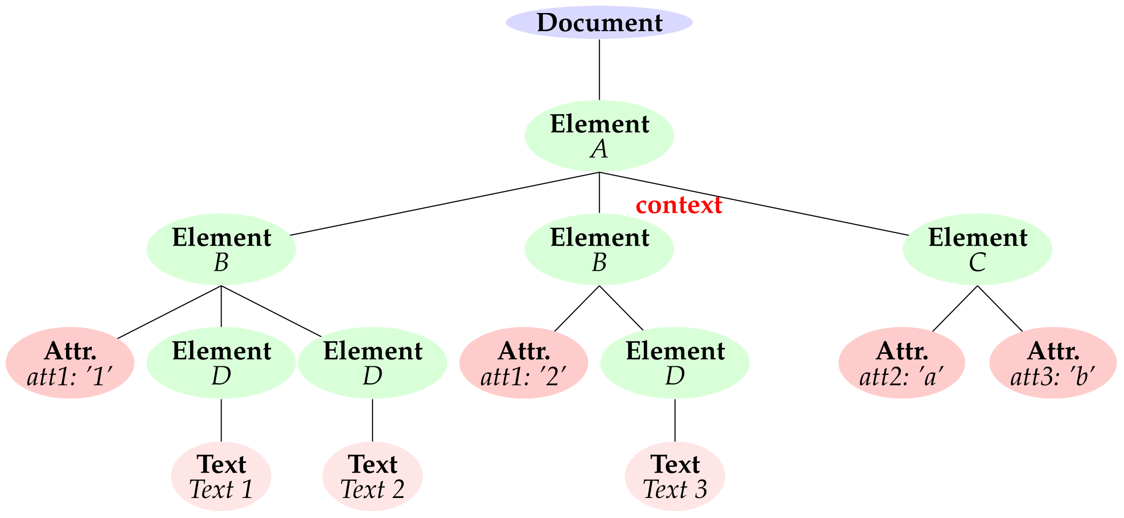

Figure 7: Evaluation of `@att1` with context node `B[2]`

order. In some cases, it is the *reverse document order*. The rule can be simply remembered as: for *forward* axes, positions follow the document order; for *backward* axes (cf. Table 1), they are in reverse order. One can also see that they correspond to how they are "naturally" visited following the navigation from the context node.

An axis is always interpreted with respect to the context node. It may happen that the axis cannot be satisfied, because of some incompatibility between the kind of the context node and the axis. An empty node set is then returned. The cases of such "impossible" moves are the following:

- When the context node is a *document node*: `parent`, `attribute`, `ancestor`, `following-sibling`, `preceding`, `preceding-sibling`.
- When the context node is an *attribute node*: `child`, `attribute`, `descendant`, `following-sibling`, `preceding-sibling`.
- When the context node is a *text node*: `child`, `attribute`, `descendant`.

We briefly observe next a subtlety. Attributes are not considered as part of the "main" document tree in the XPath data model. An attribute node is therefore not the child of the element on which it is located. (To access them when needed, one uses the `attribute` axis.) On the other hand, the `parent` axis, applied to an attribute node, returns the element on which it is located. So, applying the path `parent::*/child::*` on an attribute node, returns a node set that does not include the node one started with.

We next detail the different axes. To be able to illustrate, we also use node tests. These will be detailed further.

3.4 Axes

Child axis. The `child` axis denotes the **Element** or **Text** children of the context node. This is the default axis, used when the axis part of a step if not specified. So, `child::D` is in fact equivalent to `D`. See Figure 8.

Table 1: XPath axes

`child`	(*default axis*)
`parent`	Parent node.
`attribute`	Attribute nodes.
`descendant`	Descendants, excluding the node itself.
`descendant-or-self`	Descendants, including the node itself.
`ancestor`	Ancestors, excluding the node itself. **Backward** axis.
`ancestor-or-self`	Ancestors, including the node itself. **Backward** axis.
`following`	Following nodes in *document order* (except descendants).
`following-sibling`	Following siblings in *document order*.
`preceding`	Preceding nodes in *document order* (except ancestors). **Backward** axis.
`preceding-sibling`	Preceding siblings in *document order*. **Backward** axis.
`self`	Context node itself.

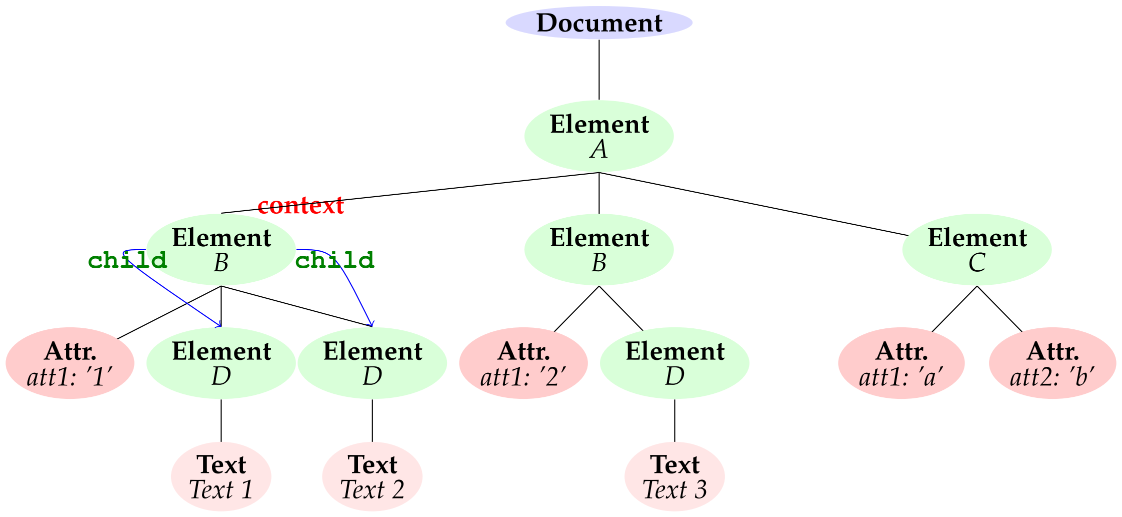

Figure 8: The `child` axis

Parent axis. The `parent` axis denotes the parent of the context node. The result is always an **Element** or a **Document** node, or an empty node-set (if the parent does not match the node test or does not satisfy a predicate). One can use as node test an element name. The node test `*` matches all names. The node test `node()` matches all node kinds. These are the standard tests on element nodes. For instance:

- if the context node is one of the `B` elements, the result of `parent::A` is the root element of our sample document; one obtains the same result with `parent::*` or `parent::node()`;
- if the context node is the root element node, then `parent::*` returns an empty set, but the path `parent::node()` returns the root node of the document.

The expression `parent::node()` (the parent of the context node) may be abbreviated as `..`.

Attribute axis. The `attribute` axis retrieves the attributes of the context node. The node test may be either the attribute name, or `@*` which matches all attribute names. So, assuming the context node is the `C` element of our example,

- `@att1` returns the attribute named `att1`;
- `@*` returns the two attributes of the context node.

Descendant axis. The `descendant` axis denotes all nodes in the subtree of the context node, *except* the **Attribute** nodes. The node test `text()` matches any **Text** node. Assume for instance that the context node is the first `B` element in the document order (Figure 9). Then :

- `descendant::node()` retrieves *all* nodes descendants of the context node, except attributes (Figure 9);
- `descendant::*` retrieves all **Element** nodes, whatever their name, which are descendant of the context node;
- `descendant::text()` retrieves all **Text** nodes, whatever their name, which are descendant of the context node.

Observe that the context node is not a descendant of itself. If one wants it in the resulting context list, one should use instead `descendant-or-self`.

Ancestor axis. The `ancestor` axis denotes all ancestor nodes of the context node. The result of `ancestor::node()`, when the context node is the first `B` element, consists of both the element root and the root node. Again, if one wants the context node to belong to the result, one should use `ancestor-or-self` instead.

Following and preceding axes. The `following` and `preceding` axes denote respectively all nodes that follow the context node in the document order, or that precede the context node, with the exception of descendant or ancestor nodes. **Attribute** nodes are *not* selected.

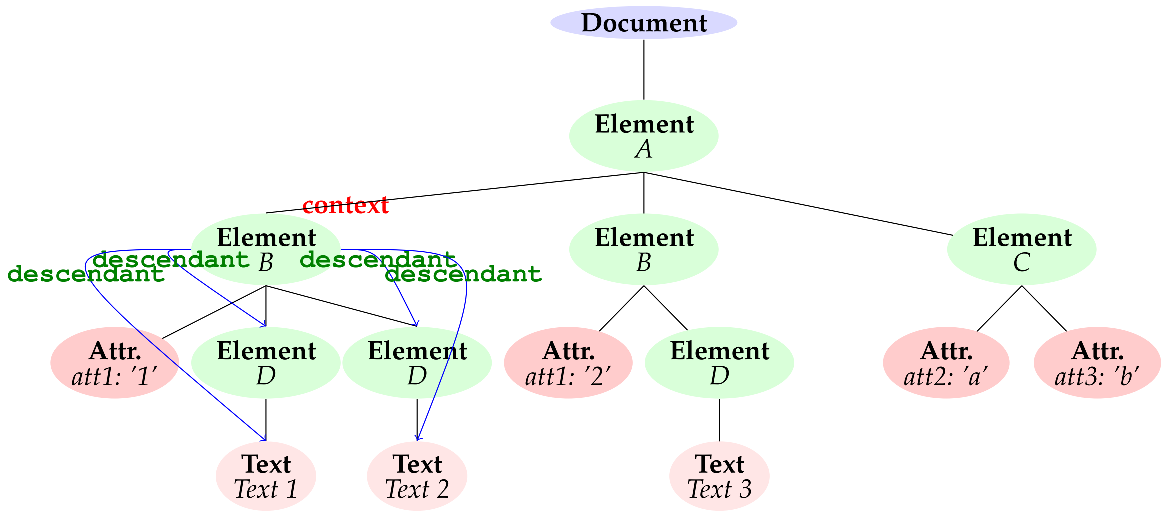

Figure 9: Result of `descendant::node()`

Sibling axes. The *siblings* of a node N are the nodes that have the same parent as N. XPath proposes two axes: `following-sibling` and `preceding-sibling`, that denote respectively the siblings that follow and precede the context node in document order. The node test that can be associated with these axes are those already described for `descendant` or `following`: a node name (for **Element**), `*` for all names, `text()` or `node()`. Note that, as usual, the sibling axes do not apply to attributes.

3.5 Node tests and abbreviations

Node tests are closely related to the kinds of the nodes. Their usage is therefore constrained to the kind of nodes returned by axis. Node tests are of the following forms:

- `node()` matches any node, except attributes;
- `text()` matches any **Text** node;
- `*` matches any named node, i.e., any **Element** node, or any **Attribute** for the `attribute` axis;
- `ns:*` or `ns:blah` match elements or attributes in the namespace bound to the prefix `ns`; the second form also imposes the exact name.

Some associations of axes and node tests are so common that XPath provides abbreviated forms. The list of abbreviations is given in Table 2.

3.6 Predicates

Predicates are optional Boolean expressions built with *tests* and Boolean connectors (`and`, `or`). Negation is expressed with the *not*() Boolean function. A *test* may take one of the following forms:

Table 2: Summary of XPath abbreviated forms

Abbreviation	Extended form
`.`	`self::node()`
`..`	`parent::node()`
`blah`	`child::blah`
`@blah`	`attribute::blah`
`a//b`	`a/descendant-or-self::node()/b`
`//a`	`/descendant-or-self::node()/a`

- an XPath expression; the semantics is that the resulting node set is nonempty;
- a comparison or a call to a Boolean function.

Predicates, the last components of an XPath expression step, provide the means to select nodes with respect to content of the document, whereas axis and node test only address the structural information. The processor first creates a sequence of nodes from the axis and the node test. The nodes in the sequence are then tested for each predicate (if any), one predicate after the other. Only those nodes for which each predicate holds are kept.

In order to understand the meaning of a precidate, we must take into account the context of the step evaluation. Recall that an XPath step is *always* evaluated with respect to the context of the previous step. This context consists of a context list, and a context node from this list. The size of the context list is known by the function *last*(), and the position of the context node in the list by *position*().

It is very common to use these functions in predicates. For instance, the following expression:

```
//B/descendant::text()[position()=1]
```

denotes the first **Text** node descendant of each node `B`. Figure 10 shows the result. Using the position is so common that when the predicates consists of a single number n, this is assumed to be an abbreviation for $position() = n$. The previous expression is therefore equivalent to:

```
//B/descendant::text()[1]
```

Expression `//B[last()]` denotes therefore the last element `B` in the document (it is an abbreviation for `//B[position()=last()]`). A predicate on a position must be carefully interpreted with respect to the context when the *position*() and *last*() functions are evaluated. It should be clear for instance that the following expressions all give different results (look at our example document, and try to convince yourself!):

1. `/descendant::B[1]/descendant::text()`,
2. `/descendant::B[1]/descendant::text()[1]`,
3. `/descendant::B/descendant::text()[1]`, and
4. `/descendant::B/D/text()[1]`.

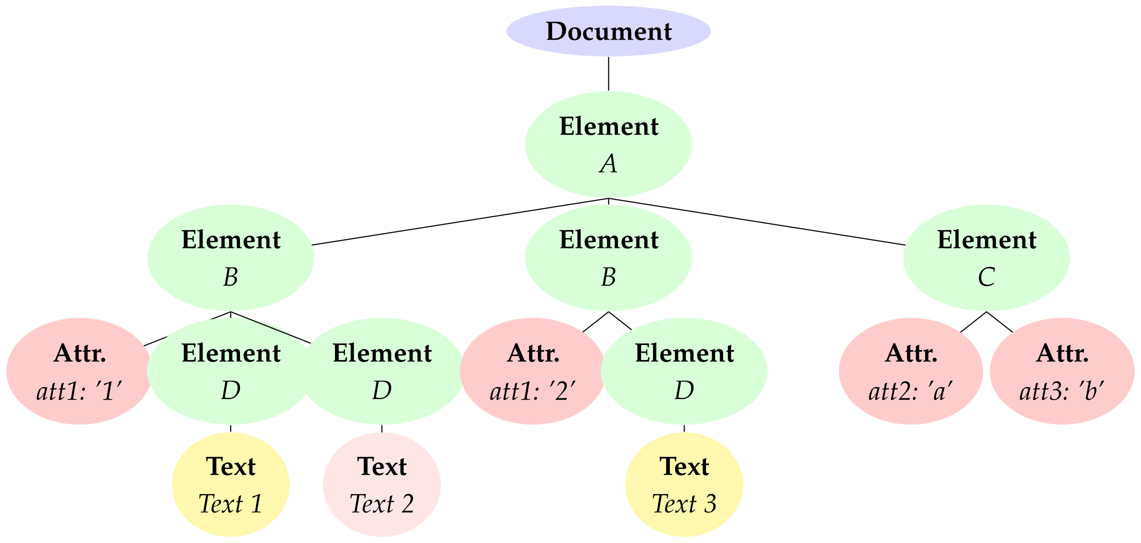

Figure 10: Result of `//B/descendant::text()[position()=1]`

Conversions in XPath

Since a predicate often consists in some test on the value or on the content of some document node(s), its evaluation may require a *conversion* to the appropriate type, as dictated by the comparator or the constant value used in the predicate expression. Consider for instance the following examples:

- `B/@att1 = 3`
- `/A/B = /A/C/@att2`
- `/A/B = /A/C`

The first case is a simple (and natural) one. It just requires a conversion of the value of the `att1` attribute to a number so that the comparison may take place. Note that this may not always be possible. For instance, if the value of the attribute is "Blah", this string cannot be coerced to be an integer and the comparison simply returns `false`. The second case is more intricate. Suppose the `/A/B` expression returns a sequence of nodes and `/A/C/@att2` returns a single attribute. Since this expression is perfectly legal in XPath, the language defines type conversion rules to interpret this comparison. Finally the last case is a comparison between two node sets. Here again, a rule that goes far beyond the traditional meaning of the equality operator is used in XPath: the result of the comparison between two node sets is true if there exists one node from the first node set and one node from the second node set for which the result of the comparison, after conversion to the appropriate data type, is true.

Thus, such comparisons are based on type and type conversion. The type system of XPath 1.0 consists of four primitive types, given in Table 3. The result of an XPath expression (including constant values) can be *explicitly* converted using the *boolean*(), *number*() and *string*() functions. There is no function for converting to a node set, since this conversion is naturally done in an *implicit* way most of the time. The conversion obeys rules that try, as far as possible, to match the natural intuition.

Table 3: The primitive types of XPath 1.0

Type	Description	Literals	Examples
Boolean	Boolean values	*none*	`true(),not($a=3)`
number	Floating-point numbers	`12,12.5`	`1 div 33`
string	Character strings	`"to",'ti'`	`concat('Hello','!')`
node set	Unordered sets of nodes	*none*	`/a/b[c=1 or @e]/d`

Conversion to a Boolean

Here are the rules for *converting to a Boolean*:

- A number is true if it is neither 0 nor *NaN*. (*NaN* stands for *Not a Number*. It is a value of the number type representing an undefined or unrepresentable value.)
- A string is true if its length is not 0.
- A node set is true if it is not empty.

An important conversion rule is the one that states that a node set is true if it is nonempty. Consider the following two examples:

- `//B[@att1=1]`: all nodes `B` having an attribute `att1` with value 1;
- `//B[@att1]`: all nodes `B` having an attribute named `att1`.

In this last example, `@att1` is an XPath expression whose result is a node set which is either empty or contains a single node, the `att1` attribute. Found in a predicate, it is converted to a Boolean. If, for a `B` node, the node set resulting from `@att1` is nonempty (the current context node has an `att1` attribute), the set is converted to the Boolean `true`.

Converting a node set to a string

Here are the rules for *converting a node set to a string*:

- The string value of an element or document node is the concatenation of the character data in all text nodes below.
- The string value of a text node is its character data.
- The string value of an attribute node is the attribute value.
- The string value of a node set is the string value of its first item in document order.[3]

These rules are illustrated by the following examples, based on the document of Figure 11.

- `boolean(/a/b)` is true;

[3]This behavior is specific to XPath 1.0. In XPath 2.0, it is an error to cast a sequence of more than one item to a string.

```
<a toto="3">
  <b titi='tutu'><c /></b>
  <d>tata</d>
</a>
```

Figure 11: XML file illustrating types conversion

- `boolean(/a/e)` is false;
- `string(/)` is `"tata"` (assuming all whitespace-only text nodes are stripped);
- `string(/a/@toto)` is `"3"`;
- `string(/a/*)` evaluates to the empty string in XPath 1.0; it raises an error in XPath 2.0.

This concludes this presentation of the essential principles of XPath. All the material presented so far is valid for XPath 1.0 which is the specification that is most commonly implemented nowadays. Some features specific to XPath 2.0 are introduced below. Note also that the expressiveness of XPath is extended with many *functions* that provide ad-hoc computations. For a large part, these functions are standardized and now belong to the XQuery specification. XML systems often add their own built-on functions, and the ability to create new ones. Chapter **??**, devoted to the EXIST system, gives a list of the most useful ones.

3.7 XPath 2.0

We briefly mention here the most important extensions that the XPath 2.0 language adds to XPath 1.0; since XPath 2.0 is a subset of XQuery, all of these are usable in XQuery:

- **Improved data model**, tightly associated with XML Schema. XPath 2.0 fully follows the XQuery data model presented earlier, including schema annotations and sequences (the semantics of simple path expressions remain the same, however; in particular the result of a path expression does not contain duplicate nodes, and is sorted in document order).
- **More expressive** language features, especially allowing to compute the intersection or set difference of a path operation (respectively, `intersect` and `except`), to branch depending on the result of a condition (`if(...) then ... else ...`), and to iterate over a sequence (`for ... return`, `some ... satisfies` and `every ... satisfies` expressions). The `for ... return` expression is a restriction of the more general XQuery FLWOR expression. Here is a showcase of some of these new capabilities of the language:

  ```
  //a//b intersect //a//c
  if(/a/b) then /a/c else /a/d
  for $x in //a return ($x,$x/..)
  //a[some $x in * satisfies $x = //b]
  ```

- **More precise** operators for value comparisons: `eq`, `ne` or `le` behave similarly as `=`, `!=` and `<=`, except they can only be applied to atomic values, not sequences of length greater than one. In the presence of schema annotations, comparison behaves accordingly to the data types of the operands. A new `is` operator allows testing node identity.
- **Ease-of-use** with many new built-in functions, including regular expression matching, date and time manipulation, extraction of distinct values in a sequence, etc.

XPath 2.0 also introduce new node tests:

item() any node or atomic value;

element() any element node;

element(author) any element named `author`;

element(*, xs:person) any element of type `xs:person`;

attribute() any attribute.

Finally, XPath 2.0 also permits nested paths expressions: any expression that returns a sequence of nodes can be used as a step. The following expression is for instance valid in XPath 2.0, but not in XPath 1.0.

```
/book/(author | editor)/name
```

4 FLWOR expressions in XQuery

We delve in this section in more detail into the fundamental aspect of XQuery, namely FLWOR expressions. As already mentioned, FLWOR queries are very close, syntactically and semantically, to SQL queries formed with **select**, **from**, **where** and **order by**. A major difference is that the output of a SQL queries is limited to the creation of flat tuples, whereas XQuery is able to nest query results in order to create complex documents with hierarchical structure.

In its simplest form, a FLWOR expression provides just an alternative to XPath expressions. For instance:

```
let $year:=1960
for $a in doc('Spider-Man.xml')//actor
where $a/birth_date >= $year
return $a/last_name
```

is equivalent to the XPath expression `//actor[birth_date>=1960]/last_name`.

Actually FLWOR expressions are much more expressive and, in general, they cannot be rewritten simply with XPath. Let us now examine in turn the clauses **for**, **let**, **where** and **return**. The use of **order by** is straightforward: it allows for the ordering of the sequence processed by the **return** clause, in the same way as the SQL keyword of the same name; the ascending or descending character of the order is specified with the **ascending** (default behavior) or **descending** keywords following the sort criterion.

4.1 Defining variables: the for and let clauses

A FLWOR expression starts with an arbitrary (non-zero) number of **for** and **left** clauses, in whatever order. A **for** clause defines a variable that ranges over a sequence. The sequence may be obtained by many means. Most commonly one uses the result of an XPath expression and the sequence often consists of nodes with similar structure. However nothing prevents a variable to range over a heterogeneous sequence that mixes values and nodes of completely unrelated structures. The following variant of the previous query is perfectly legal:

```
for $a in doc('Spider-Man.xml')//*
where $a/birth_date >= 1960
return $a/last_name
```

Note that `$a` now ranges over *all* the element nodes of the document. The semantics of XQuery states that the result is instantiated only for those nodes which feature both a `birth_date` and a `last_name`. If only `actor` nodes have both, the two are equivalent. However, this second query is typically less efficient, in particular if many nodes have one of the two and not the other.

The range of a **for** clause can also be a sequence of values, as in:

```
for $i in (1 to 10) return $i
```

As all loops in any language, **for** clauses can be nested:

```
for $i in (1 to 10) return
  for $j in (1 to 2) return $i * $j
```

The expression above realizes a *cross product* of the two input sequences. The bindings generated by these expressions consist of all the possible pairs of values. XQuery allows a more concise syntactic variant:

```
for $i in (1 to 10), $j in (1 to 2)
  return $i * $j
```

In all cases, the result of a **for** expression is the sequence of nodes and values obtained by instantiating the content of the **return** clause. In fact, a **for** clause is just an instance of an XQuery expression that returns a sequence. As such, in can be used as the range of another sequence. The following query is valid, and enumerates the multiples of 6 from 6 to 60:

```
for $i in (for $j in (1 to 10) return $j * 2)
  return $i * 3
```

XQuery is a functional language: any expression takes as input a sequence and returns a sequence. This allows expressions to be nested in one another without restrictions.

The **let** clause is just a simple way of defining a variable and assigning a value to it. The variable just acts as a synonym for its value (which, of course, is a sequence obtained by any convenient means, ranging from literals to complex queries). The following defines `$m` to be a shorthand for the root element of the *Spider-Man.xml* document.

```
let $m := doc('movies/Spider-Man.xml')/movie
return $m/director/last_name
```

Once defined, a variable can be used as its value. In the following example, the **let** clause could easily be avoided. In general, **let** is essentially a convenient way of referring to a value.

```
let $m := doc('movies/Spider-Man.xml')/movie
for $a in $m/actor
return $a/last_name
```

The scope of a variable is that of the FLWOR expression where it is defined. Since XQuery is a pure functional language, variables cannot be redefined or updated within their scope (the same rule holds for XSLT). They are in effect *constants*. This yields sometimes strange behavior, as shown by the following example:

```
let $j := 0
for $i in (1 to 4)
  let $j := $j + $i
return $j
```

One might expect that `$j` works like an accumulator which stores successively the values $(1, 1+2, 1+2+3, 1+2+3+4)$. But `$j` instead is redefined at each iteration of the **for** loop, and the resulting sequence is simply $(1, 2, 3, 4)$.

One must consider XQuery variables, just like XSLT variables, as references to values, and not as storage unit whose content can be accessed and replaced. There is indeed nothing like a global register holding some information shared by all expressions. The XQuery user must comply to the functional spirit of the language, and design its operations as trees of expressions that receive and transmit sequences, without any form of side effect. The sequence $(1, 1+2, 1+2+3, 1+2+3+4)$ can be obtained by:

```
for $i in 1 to 4 return sum (1 to $i)
```

4.2 Filtering: the where clause

The optional **where** clause allows to express conditional statements. It is quite similar to its SQL counterpart. The difference lies in the much more flexible structure of XML documents, and in the impact of this flexibility on the interpretation of the **where** statement. A few examples follow. The first one retrieves titles of films in the *movies* collection that are directed by Woody Allen, in lexicographic order.

```
for $m in collection("movies")/movie
where $m/director/last_name='Allen'
order by $m/title
```

```
return $m/title
```

This first example resembles the use of **where** in SQL. Variable `$m` ranges over the collection of movies, and each movie is selected if and only if its (unique) director is named Allen.

A first comment is that, at least in the absence of schema, nothing guarantees that the path

```
movie/director/last_name
```

is always found in in the *movies* collection. In a relational database context, data always complies to a known schema, and the query parser is always able to determine whether a query matches the schema or not, in which case an error (with explanatory messages) is produced. This is no longer systematically true in an XML database context. If the schema is unknown, the parser accepts any syntactically correct expression and attempts to match the paths in the query with the documents found in the scope of the query. If a path does not exist, then this results either in an evaluation to `false` (in the **where** clause) or an empty result (in the **return** clause). A downside of this flexibility, from the user point of view, is that mistakes in query expressions will not be rejected by the parser.

Here is another example which is only a small restatement of the previous one. We are looking for movies featuring Kirsten Dunst as an actress.

```
for $m in collection("movies")/movie
where $m/actor/last_name='Dunst'
order by $m/title
return $m/title
```

The query is syntactically correct and delivers the expected result. The subtle point here is that the path `$m/actor/last_name` returns a sequence of nodes (the list of actors in a movie), which is compared to a single value ("Dunst"). This is a specific example for the more general rule for evaluating comparison operators between two sequences: if *at least* one successful matching is found between one element of the left sequence and one element of the right one, then it evaluates to `true`, else to `false`. For our example, this can be stated as: "return those movies for which at least one of the actor names is 'Dunst'."

4.3 The return clause

The **return** clause is a mandatory part of a FLWOR expression, and always comes last. It is instantiated once for each binding of the variable in the **for** clause that passed the **where** test. The body of **return** may include arbitrary XQuery expressions, but often contain literal XML fragments that serve to structure the output of the query. Inside these XML fragments, XQuery expressions must be surrounded with braces so that the parser can identify them. Actually, nesting expressions in a **return** clause is the only means of creating non-flat results, and so, complex XML documents. The following examples shows how to output a textual representation of a movie featuring Kirsten Dunst.

A first loop outputs the information that functionally depends on each movie.

```
for $m in collection("movies")/movie
```

```
let $d := $m/director
where $m/actor/last_name='Dunst'
return
  <div>{
  $m/title/text(), ' directed by ',
        $d/first_name/text(), ' ', $d/last_name/text()
  }</div>
```

As it appears inside a literal element, the sequence inside the curly braces is interpreted as a sequence of nodes to be inserted inside the element. Atomic values (strings, numbers, etc.) are converted into text nodes containing this value, and adjacent text nodes are merged. This notation facilitates the production of text mixing literal and dynamic values. The query returns the following result:

```
<div>Marie Antoinette, directed by Sofia Coppola</div>
<div>Spider-Man, directed by Sam Raimi</div>
```

Now we need to add the list of actors. This requires a second FLWOR expression, inside the **return** clause of the first one.

```
for $m in collection("movies")/movie
let $d := $m/director
where $m/actor/last_name='Dunst'
return
  <div>{
   $m/title/text(), ' directed by ',
        $d/first_name/text(),  $d/last_name/text()}, with
    <ol>{
      for $a in $m/actor
        return <li>{concat($a/first_name, ' ', $a/last_name,
                 ' as ',  $a/role)}</li>
    }</ol>
  </div>
```

XQuery comes equipped with a large set of functions, namely all functions from XPath 2.0 (see Chapter **??** on EXIST for a short list). The above query uses *concat*(), as an alternative of the merging of text nodes used previously. One obtains finally the following output:

```
<div>Marie Antoinette, directed by Sofia Coppola, with
<ol>
<li>Kirsten Dunst as Marie Antoinette</li>
<li>Jason Schwartzman as Louis XVI</li>
</ol>
</div>

<div>Spider-Man, directed by Sam Raimi, with
<ol>
<li>Kirsten Dunst as Mary Jane Watson</li>
<li>Tobey Maguire as Spider-Man / Peter Parker</li>
```

```
<li>Willem Dafoe as Green Goblin / Norman Osborn</li>
</ol>
</div>
```

4.4 Advanced features of XQuery

In addition to FLWOR expressions, a number of aspects of the XQuery language are worth mentioning, some of which inherited from XPath 2.0.

Distinct values from a sequence can be gathered in another sequence with the help of the XPath 2.0 function *distinct-values*(). (This loses identity and order.) This is useful to implement grouping ï£¡ la SQL **group by**. For instance, the query “Return each publisher with their average book price” can be expressed as:

```
for $p in
  distinct-values(doc("bib.xml")//publisher)
  let $a :=
   avg(doc("bib.xml")//book[publisher=$p]/price)
  return
    <publisher>
      <name>{ $p/text() }</name>
      <avgprice>{ $a }</avgprice>
    </publisher>
```

The **if-then-else** branching feature of XPath 2.0 is also often useful, as in the following example that extracts some information about published resources, depending on their kind:

```
for $h in doc("library.xml")//publication
return
  <publication>
    { $h/title,
         if ($h/@type = "journal")
         then $h/editor
         else $h/author }
  </publication>
```

The existential and universal quantifier expressions from XPath can be used to express such queries as “Get the document that mention sailing and windsurfing activities” or “Get the document where each paragraph talks about sailing“.

```
for $b in doc("bib.xml")//book
where some $p in $b//paragraph
   satisfies (contains($p,"sailing")
               and contains($p,"windsurfing"))
return $b/title
```

```
for $b in doc("bib.xml")//book
```

```
where every $p in $b//paragraph
          satisfies contains($p,"sailing")
return $b/title
```

Finally, it is possible to define functions in XQuery. Such functions may be recursive. This turns XQuery into a full-fletched, Turing-complete, programming language, which is a major departure from the limited expressive power of a language like XPath. The following example shows how to define and use a function computing the factorial of a positive integer. This example also illustrates the use of primitive XML Schema types, and the fact that XQuery programs need not contain FLWOR expressions.

```
declare namespace my="urn:local";
declare namespace xs="http://www.w3.org/2001/XMLSchema";

declare function my:factorial($n as xs:integer)
  as xs:integer {
  if ($n le 1) then
    1
  else
    $n * my:factorial($n - 1)
};

my:factorial(10)
```

We end here this practical introduction to the XPath and XQuery languages. The following section explores the theoretical foundations of the XPath language.

5 XPath foundations

The main role of XPath is the selection of nodes in a tree. Its semantics is defined as some form of guided navigation in the tree browsing for particular nodes on the way. The language is rather elegant and avoids the use of explicit variables. This should be contrasted with a language such as first-order logic (FO for short), elegant in a different way, that is built around the notion of variable. In this section, we highlight a surprisingly deep connection between the navigational core of XPath 1.0, called in the following navigational XPath, and a fragment of FO, namely, FO limited to using at most two variables. We shall also mention other results that highlight the connection between various fragments of XPath 1.0 and 2.0 and FO.

These connections are best seen with an alternative semantics of XPath that proceeds bottom-up, i.e., starting from the leaves of the XPath expression and moving to its root. The "official" semantics that we previously presented suggests simple top-down implementations that turn out to be very inefficient on some queries. Indeed, the first XPath implementations that followed too closely the specification were running on some queries in time that was exponential in the size of the tree. To give an intuition of the issues, we present an example of such a query. Consider the document `<a><d/><d/></a>` and the sequence of XPath "pathological" expressions:

$pathos_0$	`/a/d`
$pathos_1$	`/a/d/parent::a/d`
$pathos_2$	`/a/d/parent::a/d/parent::a/d`
…	
$pathos_i$	`/a/d(/parent::a/d)`^{i}

A naïve evaluation of these queries that follows closely the top-down semantics we discussed has exponential running time: each addition of a navigation "up" and "down" doubles the time of the evaluation. Important improvements in algorithms for evaluating XPath are now fixing these issues. Indeed it is now known that the complexity of XPath is PTIME for all queries and good XPath processors do not run in exponential time for any query.

The problem with the evaluation of the previous query comes from an incorrect interpretation of the semantics of XPath. The result of the $pathos_1$ expressions is the node set (d_1,d_2) where d_1 is the first `d` node and d_2 the second. If the node set is seen as a sequence, as in XPath 2.0, the order is that of the document because the last axis is `child`. Now, for each i, the result of $pathos_i$ is the same node set with the same order. (And no! $pathos_2$ is not (d_1,d_2,d_1,d_2) because duplicates are eliminated in node sets.)

We next, in turn, (i) introduce a formal relational view of an XML tree, (ii) specify navigational XPath, (iii) reformulate its semantics, (iv) show that this simple fragment can be evaluated in PTIME; and (v) consider connections with first-order logic.

5.1 A relational view of an XML tree

A common way to efficiently store and query an XML database is to encode it as a relational database, as described in Chapter **??**. In a similar manner, we define here a formal view of trees as relational structures, to help define the semantics of navigational XPath.

A tree T can be represented as a relational database (a finite structure in terms of logic) as follows. Each node is given a unique identifier. We have a unary relation L_l for each label l occurring in the tree. The fact $L_l(n)$ indicates that the label of node n is l. Labels stand here for both names and values, to simplify.

We shall use a relation *nodeIds* that contains the set of identifiers of nodes in T. We also have two binary relations *child* and *next-sibling*. Observe that we are using the symbol *child* to denote both the axis and the corresponding relation. We shall do so systematically for all axes. For two node identifiers n and n', $child(n,n')$ if n' is the child of n, *next-sibling*(n,n') if n' is the next sibling of n. (They have the same parent and the position of n' is that of n plus 1.) Though *next-sibling* is not an axis available in XPath 1.0, it can be simulated by the expression `following-sibling::node()[1]`.

We can define binary relations for the other axes:

- *self* is $\{(n,n) \mid nodeIds(n)\}$;
- *descendant* is the transitive closure of *child*;
- *descendant-or-self* is the union of *self* and *descendant*;
- *following-sibling* is the transitive closure of *next-sibling*;

- *following* is

  $\{(n, q) \mid \exists m \exists p \quad ancestor\text{-}or\text{-}self(n, m) \wedge following\text{-}sibling(m, p) \wedge descendant\text{-}or\text{-}self(p,q)\}$;

- *parent*, *ancestor*, *ancestor-or-self*, *preceding-sibling*, *previous-sibling*, *preceding*, are the inverses of, respectively, *child*, *descendant*, *descendant-or-self*, *next-sibling*, *following-sibling*, *following*.

Observe that this gives a formal semantics for axes. Note also that relations *nodeIds*, *child* and *next-sibling* can be constructed in one traversal of the tree and that from them the other axis relations can be constructed in PTIME, for instance using a relational engine including transitive closure.

We use the term *node-set* to denote the powerset of *nodeIds*. An element in *node-set* is thus a set of node identifiers.

5.2 Navigational XPath

We consider a fragment of XPath 1.0 that focuses exclusively on its navigational part, namely *NavXPath*. This fragment ignores such things as equalities between path expressions, positions, or aggregate functions such as *count*() or *sum*(). To be able to better highlight the connections with logic, we depart slightly from the XPath syntax. The language is still a fragment of the whole language in that each NavXPath query can easily be translated to an "official" XPath query.

NavXPath expressions are built using the grammar:

$$\begin{array}{lllllllllll} p & ::- & step & \mid & p/p & \mid & p \cup p \\ step & ::- & axis & \mid & step[q] \\ q & ::- & p & \mid & label() = l & \mid & q \wedge q & \mid & q \vee q & \mid & \neg q \end{array}$$

where

- p stands for path expression and q for *qualifier* or *filter* (we avoid the term *predicate* here that has another meaning in first-order logic); and
- *axis* is one of the axes previously defined.

Ignoring the order first, the semantics is formally defined as follows. Since an XPath expression p may be interpreted both as an expression and as a qualifier, we have to be careful when we formally define the semantics and distinguish two semantic functions, one denoted $[.]_p$ (for path expressions) and one $[.]_q$ (for qualifiers). It is important to keep in mind that the semantic function $[.]_p$ maps a path expression p_1 to a binary relation, where $[p_1]_p(n, n')$ states that there exists a path matching p_1 from n to n'. On the other hand, the semantic function $[.]_q$ maps a qualifier q_1 to a unary relation, where $[q_1]_q(n)$ states that node n satisfies q_1. Formally, we have:

Expressions	
$[r]_p$	$:= r$ (for each axis relation r)[4]
$[step[q_1]]_p$	$:= \{(n, n') \mid [step]_p(n, n') \wedge [q_1]_q(n')\}$
$[p_1/p_2]_p$	$:= \{(n, n') \mid \exists m([p_1]_p(n, m) \wedge [p_2]_p(m, n'))\}$
$[p_1 \cup p_2]_p$	$:= [p_1]_p \cup [p_2]_p$

Qualifiers	
$[label() = l]_q$	$:= L_l$ (for each label l)
$[p_1]_q$	$:= \{n \mid \exists n'([p_1]_p(n, n'))\}$
$[q_1 \wedge q_2]_q$	$:= [q_1]_q \cap [q_2]_q$
$[q_1 \vee q_2]_q$	$:= [q_1]_q \cup [q_2]_q$
$[\neg q_1]_q$	$:= nodeIds - [q_1]_q$

A path query p_1 applied to a context node n returns a node set, that is $\{n' \mid [p_1]_p(n,n')\}$. Now let us introduce the order. Observe that the semantics so far defined specifies a *set* of nodes; this set is then ordered in document order.

Clearly, we have departed slightly from the official syntax. For instance, a query such as

$$child[label() = a][q]$$

corresponds to `child::a[q]` in standardized XPath 1.0. Observe also that the focus is on relative path expressions. It is easy to introduce absolute path expressions: one tests for the root as the (only) node without parent. It is left as an exercise to show that all queries in NavXPath can be expressed in XPath 1.0 and that the translation can be achieved in LINEAR TIME.

5.3 Evaluation

Using the bottom-up semantics we presented, we consider the evaluation of NavXPath expressions.

Again, let us start by ignoring order. As already mentioned, the *child* and *next-sibling* relations, as well as the L_l-relations for each label l, can be constructed in linear time by one traversal of the documents, using for instance for identifiers the Dewey notation (see Chapter **??**). Now *descendant* and *following-sibling* can be computed as the transitive closure of the previous ones, also in PTIME. Then one can show that each NavXPath expression can be expressed as an FO formula or as a relational algebra query over these relations. (This formula can be computed in linear time.) From this, it is easy to see that any NavXPath expression can be evaluated in PTIME in the size of the tree. In fact, one can show that it can be evaluated in PTIME in the size of the tree *and* the expression.

Now consider order. In XML, one typically chooses a node identification scheme that makes it easy to determine which node comes first in document order. So, this ordering phase can be achieved in $O(n' \cdot \log(n'))$ where n' is the size of the result. Remember that the result of a NavXPath expression is a subset of the nodes of the original document. This whole phase is therefore achieved in $O(n \cdot \log(n))$ where n is the size of the document.

We illustrate the construction of the FO formula with an example. In the example, we use an attribute to show that their handling does not raise any particular issue. In the example, a binary relation *@a* is used for each attribute `a` occurring in the tree. Similarly, we could have a binary relation *text* for the content of text nodes.

Consider the XPath expression: `descendant::a/*[@b=5]/preceding-sibling::*`, or, in NavXPath notation:

$$q = descendant[lab() = a]/child[@b = 5]]/preceding\text{-}sibling$$

Then we have:

$$\begin{aligned} q_1(n,n_1) &\equiv descendant(n,n_1) \wedge L_a(n_1) \\ q_2(n,n_2) &\equiv \exists n_1(q_1(n,n_1) \wedge child(n_1,n_2) \wedge @b(n_2,5)) \\ q(n,n_3) &\equiv \exists n_2(q_2(n,n_2) \wedge following\text{-}sibling(n_3,n_2)) \end{aligned}$$

To see a second (and last) example, consider the pathological query $pathos_3$:

[4]Do not get confused. The r at the left of $:=$ is the axis name whereas at the right it is the axis relation.

`/a/d/parent::a/d/parent::a/d/parent::a/d`

The bottom-up construction yields:

$$\begin{aligned}
d &= \{(x',y) \mid (child(x',y) \wedge L_d(y))\} \\
parent{::}a/d &= \{(y',y) \mid \exists x'(child(x',y') \wedge L_a(x') \wedge child(x',y) \wedge L_d(y))\} \\
d/parent{::}a/d &= \{(x,y) \mid \exists y'(child(x,y') \wedge L_d(y') \wedge \exists x'(child(x',y') \wedge L_a(x') \wedge child(x',y) \wedge L_d(y)))\} \\
&\ldots
\end{aligned}$$

Observe that we are in a polynomial growth even without using fancy relational query optimization.

The bottom-up semantics specifies a PTIME evaluation algorithm.[5] Of course the resulting algorithm is rather inefficient and the state of the art in XPath processing does much better. Furthermore, we treated only the navigational part of XPath. It turns out that using clever, typically top-down, algorithms, one can process any XPath query in PTIME both in the size of the tree but also of the query.

5.4 Expressiveness and first-order logic

In this section, focusing on NavXPath, we present surprising connections with first-order logic (FO) as well as stress differences. The binary predicates (relations) allowed in the logic are all axis relations: *child*, *descendant*, *following-sibling*, etc.

The translation of NavXPath to FO is straightforward based on the semantics we presented for NavXPath. In fact, it turns out that NavXPath queries correspond to some extend to FO^2. The logic FO^2 is FO limited to two variables, say x and y, which may be reused in different existential or universal quantifiers.

Recall that a NavXPath expression can be interpreted as a binary relation mapping a node n into a set of nodes, or as a logical formula with two free variables. Also, a NavXPath qualifier is interpreted as a Boolean function and so can be interpreted by a first-order logic formula with only one free variable. For instance, consider the path expression `d/parent::a/d`. Now think about it as a qualifier. Although it may seem we need four variables to express it in logic, one can make do with 2:

$$\begin{aligned}
d/parent{::}a/d(x) &= \exists y' \left(child(x,y') \wedge L_d(y') \wedge \exists x'(child(x',y') \wedge L_a(x') \wedge \exists y(child(x',y) \wedge L_d(y)))\right) \\
&\equiv \exists y \ \left(child(x,y\ ) \wedge L_d(y\ ) \wedge \exists x \ (child(x\ ,y\ ) \wedge L_a(x\ ) \wedge \exists y(child(x\ ,y) \wedge L_d(y)))\right)
\end{aligned}$$

Check carefully these formulas to convince yourself that they are indeed equivalent. Get a feeling why it is in general the case that we can express NavXPath qualifiers with FO^2.

The precise theorem that relates FO^2 and NavXPath is a bit intricate because, to move to XPath expressions as opposed to qualifiers, we already need variables to account for the source and target. But every FO^2 formula can be expressed in XPath and every qualifier can be expressed in FO^2.

Although translations between XPath and FO^2 exist, one may wonder about the size of the results. It is known that for some FO^2 queries, the equivalent NavXPath expressions have exponential size. For the other direction, the translation of an XPath qualifier can be done in polynomial time.

[5]This is because of the restrictions coming with NavXPath; more powerful fragments of XPath 1.0 cannot easily be evaluated with a bottom-up approach.

Since it is known that some FO queries require more than 2 variables, there are FO queries that cannot be expressed in NavXPath. For instance, the following query cannot be expressed in NavXPath: there is a path from some `a` node to a descendant `a` node that traverses only `b` nodes. Indeed, this query cannot be expressed in XPath 1.0.

5.5 Other XPath fragments

NavXPath only covers the navigational core of XPath 1.0; in particular, it is impossible to express queries about the value equalities of nodes of the tree such as

```
movie[actor/@id=director/@id]/title
```

It is possible to define a formal extension of NavXPath that adds this capability. The characterization of this language in terms of first-order logic is less clear, however. It has been shown that it is a proper subset of FO^3 over the previously mentioned relations, as well as binary relations that express value comparisons. As already mentioned, query evaluation remains PTIME in terms of data-and-query complexity.

XPath 2.0 is a much more powerful language than XPath 1.0, with the intersection and complementation operators, the iteration features, etc. Its navigational core has the same expressive power as the whole of FO. Evaluating a navigational XPath 2.0 query is, however, a PSPACE-complete problem in data-and-query complexity.

6 Further reading

XPath

XPath 1.0 is a W3C recommendation [W3C99] that was released in November 1999. The relative simplicity of the language makes the recommendation quite readable to application programmers, in contrast to other W3C recommendations that describe more involved technologies.

There exists a large number of implementations for XPath 1.0. Here are a few examples freely available for various programming languages:

libxml2: Free *C* library for parsing XML documents, supporting XPath.

java.xml.xpath: *Java* package, included with JDK versions starting from 1.5.

System.Xml.XPath: standard *.NET* classes for XPath.

XML::XPath: free *Perl* module, includes a command-line tool.

DOMXPath: *PHP* class for XPath, included in PHP5.

PyXML: free *Python* library for parsing XML documents, supporting XPath.

XPath is also directly usable for client-side programming inside all modern browsers, in JavaScript.

The W3C published in January 2007 a recommendation [W3C07a] for the XPath 2.0 language. This document is not self-contained, it refers to two additional recommendations, one for describing the data model [W3C07c], and the other to describe all operators and functions [W3C07e]. An excellent reference book to the XPath 2.0 language (and to XSLT 2.0) is [Kay08], by Michael Kay, the author of the SAXON XSLT and XQuery processor.

The large number of extensions that were brought to the language, especially in connection to XML Schema annotations, make it a much more complex language, with far fewer implementations. In essence, there are no implementations of XPath 2.0 outside of XQuery and XSLT 2.0 implementations.

XQuery

XQuery was standardized along XPath 2.0, and its recommendation [W3C07b] is also dated January 2007. In addition to the recommendations on its data model and functions, cited above, there are separate documents that describe its semantics [W3C07d] as well as its serialization features [W3C07f]. The reference information is thus spread across five documents, not counting the recommendations of XML itself, XML namespaces, and XML schemas, which does not help readability. More didactic presentations of the language can be found in [MB06, Wal07].

There are a large number of XQuery implementations, both as standalone processors and as part of a XML database management system. Among the freely available (most of which provide support for the core language, but have no support for external XML schemas), let us cite:

SAXON: in-memory Java and .NET libraries; the open-source version has no support of external XML Schemas, but it is still a very convenient tool.

GNU QEXO: a very efficient open-source processor that compiles XQuery queries into Java bytecode; does not support all features of the language.

QIZX: Java libraries for both a standalone processor and a native XML database; open and free versions have limitations.

EXIST: an open-source XML database management system, with a very user-friendly interface.

MONETDB: an in-memory column-oriented engine for both SQL and XQuery querying; among the fastest.

An interesting benchmarking of some freely available XQuery processors is [MMM08]. The W3C maintains a list of XQuery processors at
`http://www.w3.org/XML/Query/#implementations`.

XPath foundations

The literature on XPath expressiveness, complexity, and processing is quite impressive. The material of this section borrows a lot from the article "XPath Leashed" [BK08] that is an in-detail discussion of expressiveness and complexity of various fragments of XPath 1.0. Efficient algorithms for processing XPath queries are presented in [GKP05]. Another interesting survey of expressiveness and complexity results can be found in [tCM07], which is one of the few research works that look at the expressiveness of XPath 2.0. XQuery is a Turing-complete language, but its core can also be analyzed with respect to first-order logic, as is done in [BK09].

7 Exercises

Most of the following exercises address the principles of XPath or XQuery. They are intended to check your understanding of the main mechanisms involved in XML documents manipulation. These exercises must be completed by a practical experiment with an XPath/XQuery evaluator. You can refer to the list of XPath and XQuery implementations Section 6. The EXIST XML database, in particular, is simple to install and use. Chapter **??** proposes several exercises and labs with EXIST.

The exercises that follow refer to a few XPath functions whose meaning should be trivial to the reader: *count()* returns the cardinality of a node-set; *sum()* converts the nodes of a node-set in numbers, and sums them all, *name()* returns the name (label) of a node, etc. Chapter **??** gives a list of common XPath/XQuery functions.

Exercise 7.1 *Consider the XML document shown on Figure 12. We suppose that all text nodes containing only whitespace are removed from the tree.*

```
<a>
  <b><c /></b>
  <b id="3" di="7">bli <c /><c><e>bla</e></c></b>
  <d>bou</d>
</a>
```

Figure 12: Sample document for Exercise 7.1

1. *Give the result of the following XPath expressions:*
   (a) `//e/preceding::text()`
   (b) `count(//c|//b/node())`
2. *Give an XPath 1.0 expression for the following queries, and the corresponding result:*
   (a) *Sum of all attribute values.*
   (b) *Text content of the document, where every "b" is replaced by a "c" (Hint: use function translate(s, $x_1 x_2 \cdots x_n$, $y_1 y_2 \cdots y_n$) that replaces each x_i by y_i in s).*
   (c) *Name of the child of the last "c" element in the tree.*

Exercise 7.2 *Explain the difference between the following two XPath expressions:*

- `//c[position() = 1]`
- `/descendant::c[position() = 1]`

Give an example of a document for which both expressions yield a different result.

Exercise 7.3 *Explain the following expressions, and why they are not equivalent.*

- `//lecture[name='XML']`

- `//lecture[name=XML]`

Give an instance that yields the same result.

Exercise 7.4 (Node tests) *Give the appropriate combination of axis and node tests to express in XPath the following searches.*

- *select all nodes which are children of an* `A` *node, itself child of the context node;*
- *select all elements whose namespace is bound to the prefix* `xsl` *and that are children of the context node;*
- *select the root element node;*
- *select the* `B` *attribute of the context node;*
- *select all siblings of the context node, itself included (unless it is an attribute node);*
- *select all* `blah` *attributes wherever they appear in the document.*

Exercise 7.5 (Predicates) *Give the results of the following expressions when applied to our example document (Figure 3, page 7).*

1. `//B[1]//text()`*,*
2. `//B[1]//text()[1]`*,*
3. `//B//text()[1]`*, and*
4. `//B/D/text()[1]`*.*

Exercise 7.6 *For each of the following XPath expressions, explain its meaning and propose an abbreviation whenever possible.*

- `child::A/descendant::B`
- `child::*/child::B`
- `descendant-or-self::B`
- `child::B[position()=last()]`
- `following-sibling::B[1]`
- `//B[10]`
- `child::B[child::C]`
- `//B[@att1 or @att2]`
- `*[self::B or self::C]`

Exercise 7.7 (XQuery and recursion) *We get back to MathML documents. Recall that arithmetic formulas are written in prefix notation (see Exercise* **??***, page* **??***). In this exercise, we adopt the following restrictions: the only operators are* `<plus/>` *and* `<times/>`*), and these operators are binary.*

1. *Give an XQuery expression that transforms an* `apply` *expression in infix form. For instance, applied to the following document:*

```
<apply>
  <times/>
   <ci>x</ci>
   <cn>2</cn>
</apply>
```

the query returns"`x * 2`*".*

2. *Assume now that the infix expression can be expressed as a function* `eval($op, $x, $y)`*, where* `$op` *is an operation,* `$x` *and* `$y` *two operands. XQuery makes it possible to call recursively any function. Give the query that transforms a MathML expression in infix form. For instance, applied to the following document*

```
<apply>
  <times/>
   <apply>
     <plus/>
     <ci>x</ci>
     <cn>2</cn>
    </apply>
    <ci>y</ci>
</apply>
```

the query should return "`(x + 2) * y`*".*

Exercise 7.8 *Show that all NavXPath queries can be expressed in XPath 1.0 and that the transformation can be achieved in* LINEAR TIME*.*

Exercise 7.9 *At the end of Section 5, it is stated that the query "there is a path from some* `a` *node to a descendant* `a` *node that traverses only* `b` *nodes" cannot be expressed in XPath 1.0. Can you find an XPath 2.0 expression for this query?*

References

[BK08] Michael Benedikt and Christoph Koch. XPath leashed. *ACM Computing Surveys*, 41(1), 2008.

[BK09] Michael Benedikt and Christoph Koch. From XQuery to relational logics. *ACM Trans. on Database Systems*, 34(4), 2009.

[GKP05] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient algorithms for processing XPath queries. *ACM Trans. on Database Systems*, 30(2):444–491, 2005.

[Kay08] Michael Kay. *XSLT 2.0 and XPath 2.0 Programmer's Reference*. Wrox, fourth edition, May 2008.

[MB06] Jim Melton and Stephen Buxton. *Querying XML: XQuery, XPath, and SQL/XML in context*. Morgan Kaufmann, March 2006.

[MMM08] Philippe Michiels, Ioana Manolescu, and Cédric Miachon. Toward microbenchmarking XQuery. *Inf. Systems*, 33(2):182–202, 2008.

[tCM07] Balder ten Cate and Maarten Marx. Navigational XPath: calculus and algebra. *SIGMOD Record*, 36(2):19–26, 2007.

[W3C99] W3C. XML path language (XPath). `http://www.w3.org/TR/xpath/`, November 1999.

[W3C07a] W3C. XML path language (XPath) 2.0. `http://www.w3.org/TR/xpath20/`, January 2007.

[W3C07b] W3C. XQuery 1.0: An XML query language. `http://www.w3.org/TR/xquery/`, January 2007.

[W3C07c] W3C. XQuery 1.0 and XPath 2.0 data model (XDM). `http://www.w3.org/TR/xpath-datamodel/`, January 2007.

[W3C07d] W3C. XQuery 1.0 and XPath 2.0 formal semantics. `http://www.w3.org/TR/xquery-semantics/`, January 2007.

[W3C07e] W3C. XQuery 1.0 and XPath 2.0 functions and operators. `http://www.w3.org/TR/xquery-operators/`, January 2007.

[W3C07f] W3C. XSLT 2.0 and XQuery 1.0 serialization. `http://www.w3.org/TR/xslt-xquery-serialization/`, January 2007.

[Wal07] Priscilla Walmsley. *XQuery*. O'Reilly, March 2007.