XPath

December 16, 2008

XPath is an expression language which constitutes the basic means to navigate in an XML tree. XPath expressions (or *path expressions*) denote paths in a tree, using a mixture of structural information (node names, node types) and constraints on data values. A path expression is evaluated with respect to a *context node*. The *result* of a path expression is the set of terminal nodes of the paths that start from the context node and match the expression. Here are a few examples:

- `/client[name='Jones']/order/` is a path expression that denotes all the orders of clients named Jones;
- `//*[@msg="Hello world"]` is a *path expression*, retrieving all elements with a `msg` attribute set to "`Hello world`";
- `2*3` is a *literal expression*: XPath also provides a means to express operations.

The power of XPath is relatively limited in term of nodes retrieval. However it constitutes a convenient tool for describing classes of paths in XML trees, and can be associated to more powerful languages for complex navigations. XSLT is a *restructuring* or *transformation* language that relies heavily on XPath for navigation, tests and value extraction. XQuery is an extension of XPath that considerably augments its power. Mastering XPath expressions is thus a pre-requisite for these two essential XML manipulation languages.

The present chapter mostly covers version 1.0 of XPath, following the W3C recommendation published in 1999. The end of the chapter proposes a brief introduction to XPath 2.0.

1 XPath Data Model

XPath expressions operate over *XML trees*. Each node in a tree has a *type*, and possibly a *name* and/or a *value*. These concepts are important for the correct interpretation of path expressions. Here is first the list of the nodes types which can be found in an XML tree:

- **Document**: the *root node* of the XML document, denoted by "/";
- **Element**: element nodes, which correspond to the tags the serialized representation of the document;
- **Attribute**: attribute nodes, represented as children of an **Element** node;
- **Text**: text nodes, i.e., leaves of the XML tree.

Note that the XPath data model also features **ProcessingInstruction** and **Comment** node types. They are seldom used, and can be addressed just as other types so we do not consider them in the following presentation. It is worth mentioning that XPath trees ignore syntactic features which are only relevant to serialized representations. For instance entities do not appear, since they are supposed resolved at XPath evaluation time.

```
<?xml version="1.0"
    encoding="UTF-8"?>
<A>
  <B>
    <D>Text 1</D>
    <D>Text 2</D>
  </B>
  <B>
    <D>Text 3</D>
  </B>
  <C att1="2"
     att2="3"/>
</A>
```

Figure 1: An example of XML document in serialized form

Figure 1 shows a serialized representation of an XML document, and Figure 2 shows the XML tree which supports XPath expressions. It is important to keep in mind some charateristics which are common to all tree representations, and help to understand the meaning of XPath 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. Unless otherwise specified, the nodes in the result of an XPath expression are given in document order.

First 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, i.e., comments or processing instructions but they are far less common (and important).

Next, for each node in a tree, the concepts of *name* and *value* are defined as follows:

- an **Element** node has a name (i.e., the tag in the serialized representation), but no value;
- a **Text** node has a value (a character string), but no name;
- an **Attribute** node has both a name and a value.

Although an **Element** node N has no value, it has a *content*, which is the XML subtree rooted at N. The *textual value* of N is sometimes used to denote the concatenation of the values of the **Text** node which are descendant of N.

Finally keep in mind that **Attribute** nodes are special: attributes are not considered as first-class nodes in an XML tree. They must be addressed specifically, when needed.

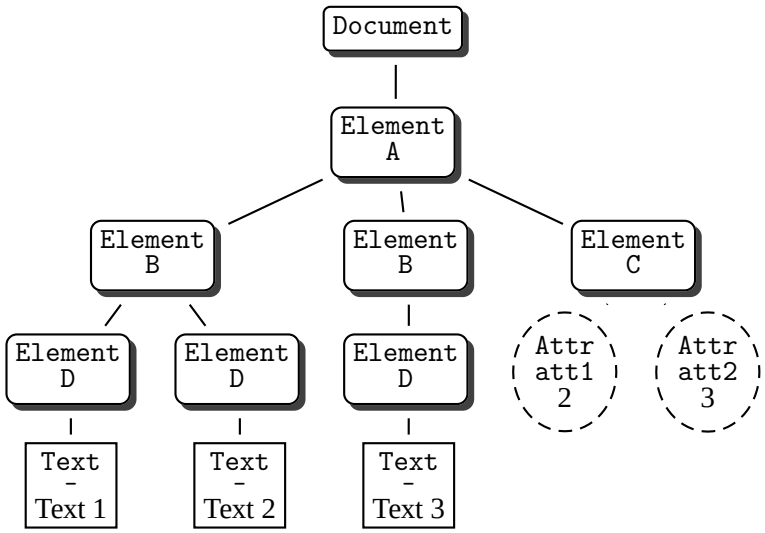

Figure 2: The tree representation of an XML document

2 Path Expressions

An XPath expression consists of *steps*, A step is always evaluated in a specific *context*

$$[< N_1, N_2, \cdots, N_n >, N_c]$$

which consists of:

- *a context list* $< N_1, N_2, \cdots, N_n >$ of nodes from the XML tree;
- *a context node* N_c belonging to the context list.

The *context length n* is a positive integer indicating the *size* of a contextual list of nodes. It can be obtained with the function `last()`. The *context node position* $c \in [1, n]$ is a positive integer indicating the *position* of the context node in the context list of nodes; it can be known by using the function `position()`.

2.1 Steps and path expressions

XPath *steps* are of the form:

$$\text{axis}::\text{node-test}[\text{P}_1]\,[\text{P}_2]\ldots[\text{P}_n]$$

Here, `axis` is an *axis name* indicating what the direction of the step in the XML tree is (*child* is the default), and `node-test` is a *node test*, indicating the kind of nodes to select. Each P_i is a *predicate*, that is, any XPath expression, evaluated as a boolean, indicating an additional condition. There may be no predicate at all.

A step is evaluated with respect to a *context*, and returns a *node list*. The following examples illustrate these concepts:

1. `child::A` (`A` in short) is a step which denotes all the **Element** children named `A` of the context node; `child` is the axis, `A` is the node test (it restricts the selected elements on their names) and there is no predicate;

2. `descendant::C[@att1='1']` is a step which denotes all the **Element** nodes named `C`, descendant of the context node, and having an **Attribute** node `att1` with value 1; note the difference between the node test which relies on structural information (node's names) and the predicate which relies on values;

3. `parent::*[B]` is a step which denotes all the **Element** nodes, whatever their name (node step `*`), parent of the context node, and having an **Element** child named `B`; the predicate here checks the existence of a node.

A path expression is of the form:

$$[/]\texttt{step}_1/\texttt{step2}/.../\texttt{step}_n$$

An expression that begins with / is an *absolute* path expression: the context of the first step is in that case the *root node*. A path that does not begin with / is a *relative* path expression. For a relative 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 expression can be found in *templates*: the XSLT execution model ensures that the a 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. `./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;

3. `/A/B/@att1[.>2]` denotes all the **Attribute** nodes `@att1` whose value is greater than 2.

In the last expression above, `.` is a special step (actually, an abbreviation of the step `self::node()`, which refers to the context node. Thus, `./dummy` means the same thing as `dummy`.

2.2 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 is *absolute*, the context is a singleton, the root of the XML tree; else (*relative* paths) the context is defined by the environment;
- For $i > 1$: if $\mathcal{N}_i =< N_1, N_2, \cdots, N_n >$ is the result of step $\texttt{step}_{i-1}$, $\texttt{step}_i$ 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 3).

The first step, `A`, is evaluated with respect to this context, and results in the element node which, in turns, becomes the context node for the second step (Figure 4).

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`.

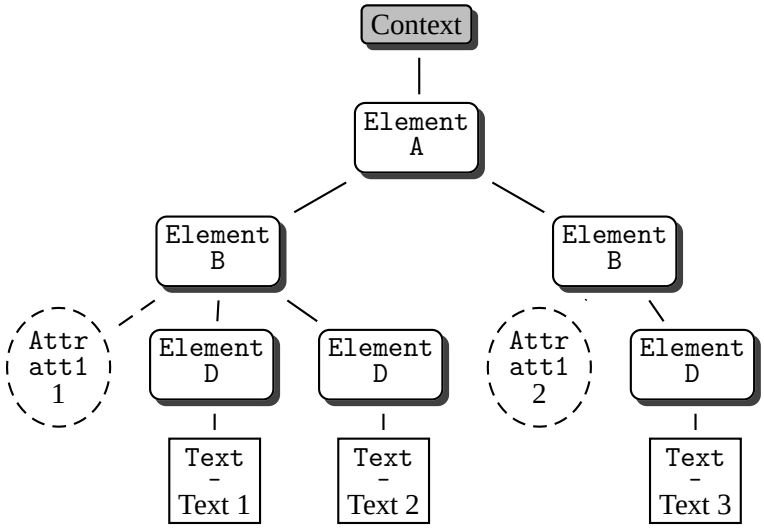

Figure 3: First step of evaluation

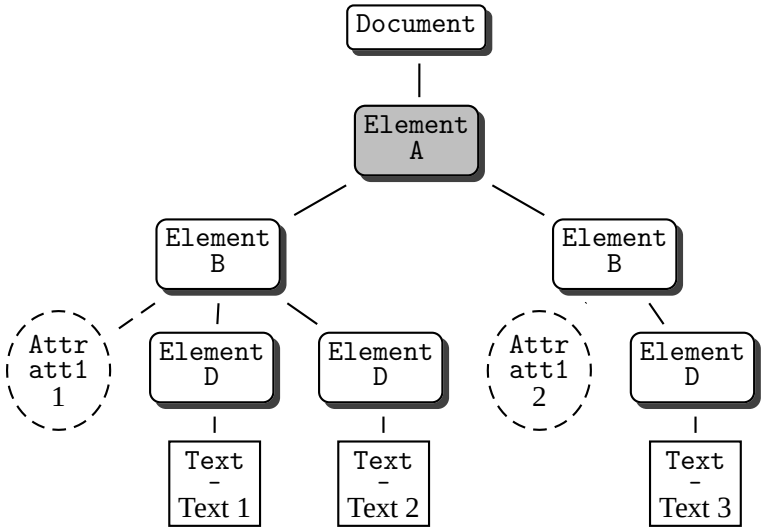

Figure 4: Second step of evaluation

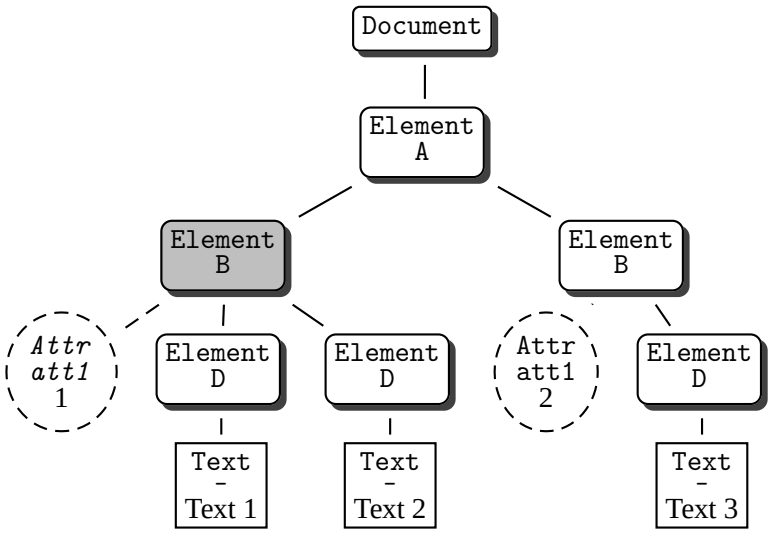

Figure 5: Evaluation of `@att1` with context node `B[1]`

1. taking the first element `B` child of `A` as context node, one obtains its attribute `@att1` (Figure 5);
2. taking the second element `B` child of `A` as context node, one obtains its attribute `@att1` (Figure 6).

The final result is the sequence of nodes union of all the results of the last step, `@att1`.

2.3 Axes and node tests

The axis of a step determines, with respect to the context node, the subset of of the document nodes for which both the node test and the predicates (if any) must be evaluated. In a predicate following a step, the context size is initialized to the number of nodes matched by both the axis and the node test. The predicate is evaluated for each different node matched with a different context position.

Table 1 gives the list of all the XPath axis. The axis also implies an order on this node set, which follows in most cases the document order, but may sometimes be the *reverse document order*. The rule can be simply stated as: for *forward* axes, positions follow the document order; for *backward* axes (cf. Table 1), they are in reverse order.

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 type of the context node and the axis. An empty node set is returned. This happens in the following cases:

- for `parent`, `attribute`, `ancestor`, `ancestor-or-self`, `following-sibling`, `preceding`, `preceding-sibling` when the context node is a *document node*;
- for `child`, `attribute`, `descendant`, `descendant-or-self`, `following-sibling`, `preceding-sibling` when the context node is an *attribute node*;
- for `child`, `attribute`, `descendant`, `descendant-or-self` when the context node is a *text node*;

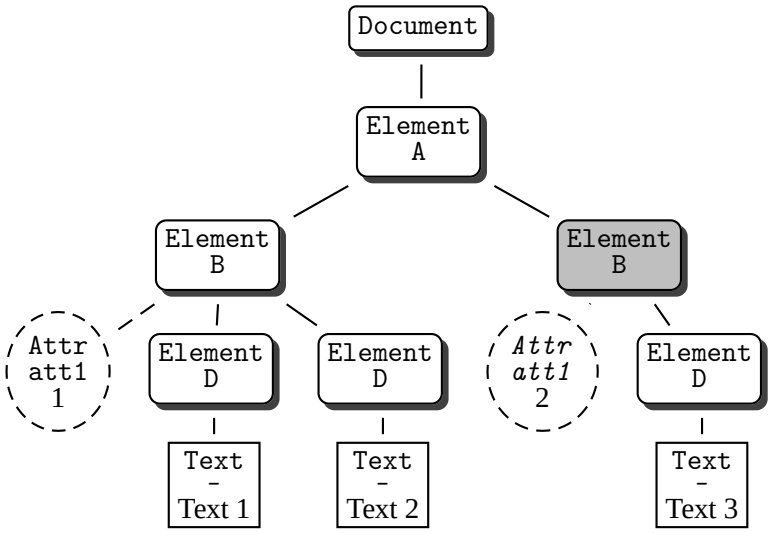

Figure 6: Evaluation of `@att1` with context node `B[2]`

`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.

Table 1: XPath axis

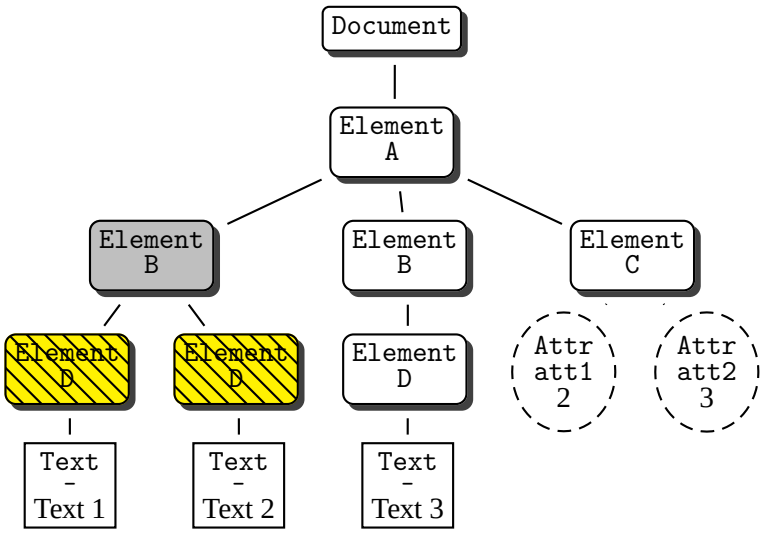

Figure 7: The `child` axis

An attribute node has a parent (the element on which it is located), but an attribute node is not one of the children of its parent. Attributes are not considered as part of the "main" document tree in the XPath data model. They must be explicitly accessed when needed, using the `attribute` axis.

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. Figure 7 shows the result of `child::D` (which is equivalent to `D`).

Parent axis

The `parent` axis denotes the parent of the context node. The result is always a **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).

The node test is either an element name, or `*` which matches all names, or `node()` which matches all node types. For instance:

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

`..` is an abbreviation for `parent::node()`: the parent of the context node, whatever its type, if it exists (cf Figure 8).

Attribute axis

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

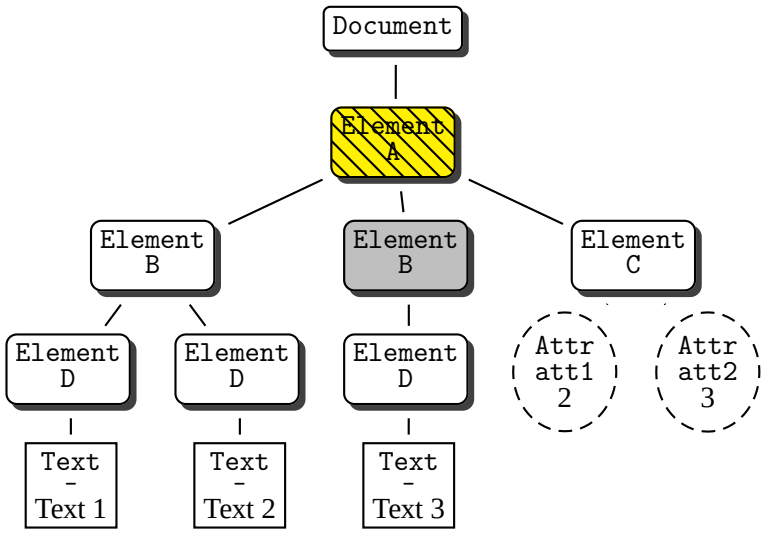

Figure 8: Result of `parent::node()`, or `..`

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

Descendant axis

The `descendant` axis denotes all the descendant nodes, *except* the **Attribute** nodes. The node test is either the node name (for **Element** nodes), or `*` (any **Element** node) or `text()` (any **Text** node) or `node()` (all nodes).

Assume for instance that the context node is the first `B` element in the document order (Figure 9). Then :

- `descendant::node()` retrieves *all* the nodes which descendant of the context node, except the attributes (which do not match the `node()` node test) (Figure 9);
- `descendant::*` retrieves all the **Element** nodes, whatever their name, which are descendant of the context node;
- `descendant::text()` retrieves all the **Text** nodes, whatever their name, which are descendant of the context node.

The context node does *not* belong to the result: use `descendant-or-self` instead.

Ancestor axis

The `ancestor` axis denotes all the ancestor nodes of the context node. The node test is either the node name (for **Element** nodes), or `node()` (any **Element** node, and the **Document** root node). Figure 10 shows the result of `ancestor::node()` when the context node is the first `B` element. The context node does *not* belong to the result: use `ancestor-or-self` instead.

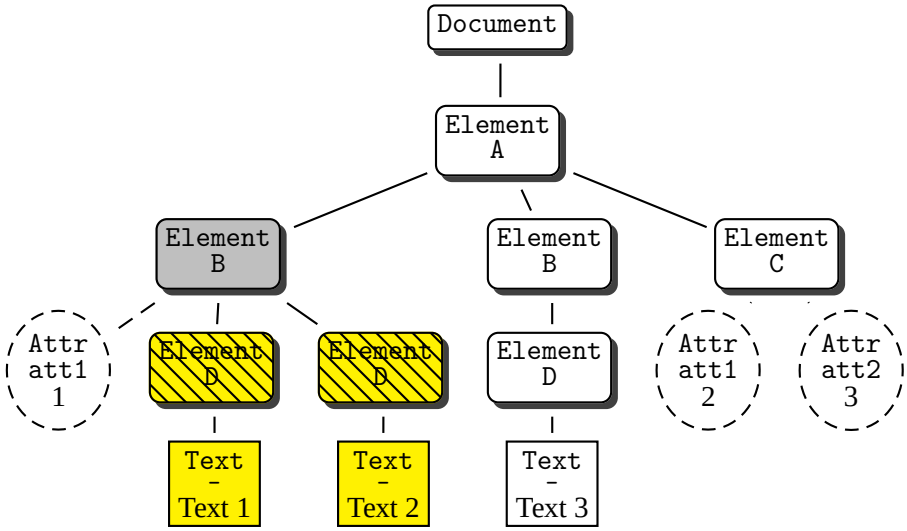

Figure 9: Result of `descendant::node()`

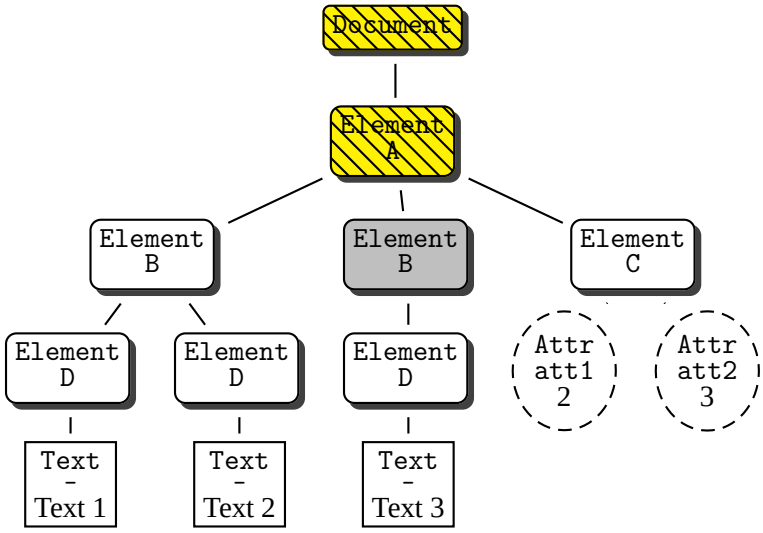

Figure 10: Result of `ancestor::node()`

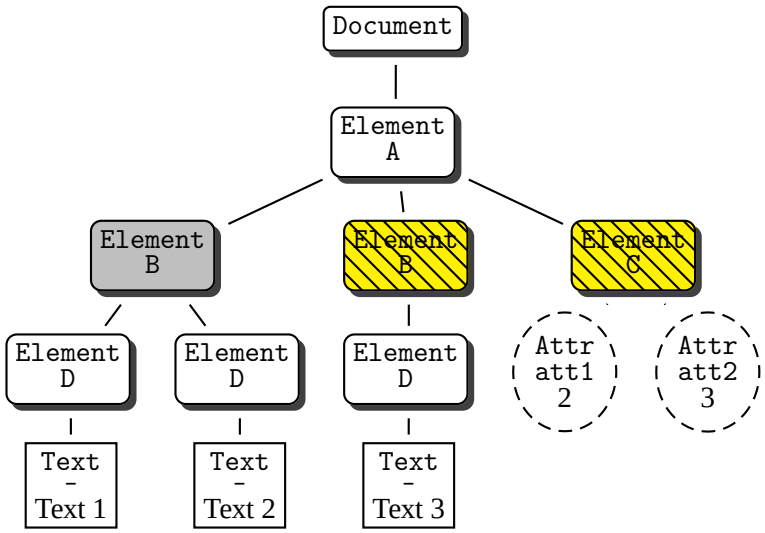

Figure 11: **Result of** `following-sibling::*`

Following and precedeing axis

The `following` and `preceding` axis denote respectively all the nodes that follow the context node in the document order, or that precede the context node. **Attribute** nodes are *not* selected. The node test is either the node name, `*` `text()` or `node()`.

Sibling axis

The *siblings* of a node *N* are the node which have the same parent as *N*. XPath proposes two axis: `following-sibling` and `preceding-sibling`, that denote respectively the siblings that follow and precede the context node in the document order. The node test that can be associated with these axis 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 axis do not apply to attributes. Figure 11 shows the result of `following-sibling`.

Node tests and abbreviations

As shown by the examples given above, node test are closely associated to the axis in XPath expressions, and they determine together a precise subset of the node from the context node. A node test has one of the following forms:

- `node()` any node, except attributes.
- `text()` any **Text** node.
- `*` is a node test that match any named node, i.e., any **Element** node, or any **Attribute** for the `attribute` axis.
- `ns:*` any element or attribute in the namespace bound to the prefix `ns`;
- `ns:bah` any element or attribute whose name is `ns:bah`

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

Figure 12: Summary of abbreviations

Many examples of node test have already been given, and their interpretation should now be clear to the reader. Here is a list that illustrate the many possible combinations of axis and node tests, and their meaning.

- `a/node()` selects all nodes which are children of a `a` node, itself child of the context node;
- `xsl:*` selects all elements whose namespace is `ns` and that are children of the context node;
- `/*` selects the top-level element node;
- `@b` selects the `b` attribute of the context node;
- `../*` selects all siblings of the context node, itself included (unless it is an attribute node);
- `//@blah` selects all `blah` attributes wherever their position in the document;

Some associations axis/node test are so common that XPath provides and abbreviated form. The list of abbreviations is given in Table 12.

2.4 Predicates

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

- an XPath expression, whose result is converted to a Boolean;
- a comparison or a call to a Boolean function.

Precidates constitute the mean to select nodes with respect to content of the document, whereas axis and node test only address the structural information. Predicates are the last component of a step considered by the XPath processor at evaluation time. The processor first creates a sequence of node from the axis and the node tests, and then checks which nodes in the sequence do match the predicate(s). If a step is of the form `axis::node-test[P]`.

- First `axis::node-test` is evaluated: one obtains an intermediate result I
- Second, for each node in I, P is evaluated: the step result consists of those nodes in I for which P is true.

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','!')`
nodeset	Unordered sets of nodes	*none*	`/a/b[c=1 or @e]/d`

Table 2: The primitive types of XPath

Predicates are also the only optional part of expression steps.

Since a predicate often consists in some test on the value or on the content of some document node(s), its evaluation requires a *conversion* to the appropriate type, as dictated by the comparator and/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 (note that this may not always be possible).

The second case is more intricate: the `/A/B` expression is likely to return a sequence of nodes, while `/A/C/@att2` is an attribute value. Since this expression is perfectly legal in XPath, the language defines rules to interpret this comparison.

Finally the last case is a comparison between two node sets. Here agin, a rule that goes far beyon the traditional meaning of the equility operator is used in XPath.

The type system of XPath consists of four primitive types, given in Table 2. The result of an XPath expressions (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 to rules which try, as far as possible, to match the natural intuition.

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

- a number is true if it is neither 0 nor *NaN*.
- a string is true if its length is not 0.
- a nodeset is true if it is not empty.

Essentially, any value which is not null nor empty is interpreted as `true`, else it is `false`. An important conversion rule which sometimes gives rise to weird expressions is the one that states that an empty node set is false. Consider the following two examples!

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

Indeed, `@att1` is an XPath expression whose result (a node set which is either empty, or contains a single node, the `att1` attribute) is converted to a Boolean.

Here are the rules for *converting a nodeset to a string*:

- The string value of a nodeset is the string value of its first item in document order.
- 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.

This rules are illustrated by the following examples, based on the document of Figure 13.

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

Figure 13: An XML file for illustrating types conversion

- `string(/)` is "tata";
- `string(/a/@toto)` is "3";
- `boolean(/a/b)` is true;
- `boolean(/a/e)`is false.

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 is known by *position*(). It is very common to use these function in predicates. So, the following expression:

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

denotes the first **Text** node descendant of each node `B`. Figure 14 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

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

all give different results (look at our example document, and try to convince yourself!).

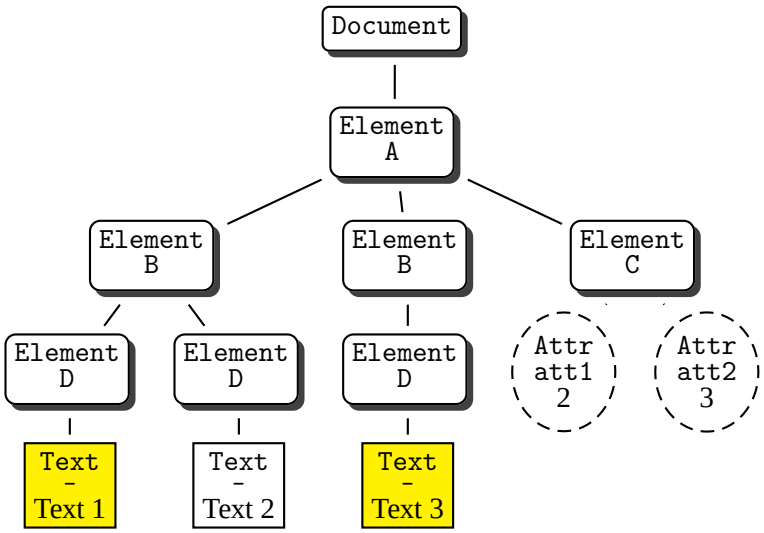

Figure 14: Result of `/A/B//text()[1]`

3 Operators and Functions

XPath proposes many operators and function to manipulate data values. These operators are mostly used in predicates, but they are also important as part of the XQuery language which inherits all the XPath language.

3.1 Operators

The following operators can be used in XPath.

- `+`, `-`, `*`, `div`, `mod` standard arithmetic operators (Example: 1+2*-3).

  Warning! `div` is used instead of the usual `/`.

- `or`, `and` boolean operators (Example: `@a and c=3`)

- `=`, `!=` equality operators. Can be used for strings, booleans or numbers.

  Warning! `//a!=3` means: there is an `a` element in the document whose string value is different from 3.

- `<`, `<=`, `>=`, `>` relational operators

  (Example: `($a<2) and ($a>0)`). *Warning!* Can only be used to compare numbers, not strings. If an XPath expression is embedded in an XML document (this will be the case with XSLT), `<` must be escaped as `<`.

- `|` union of nodesets (Example: `node()|@*`)

`$a` is a *reference* to the variable `a`. Variables can not be defined in XPath, they can only be referred to.

3.2 Node Functions

The following functions apply to node sets.

- `count($s)` returns the *number of items* in the nodeset `$s`
- `local-name($s)` returns the *name* of the first item of the nodeset $s in document order, *without* the namespace prefix; if $s is omitted, it is taken to be the context item
- `namespace-uri($s)` returns the *namespace URI* bound to the prefix of the name of the first item of the nodeset $s in document order; if $s is omitted, it is taken to be the context item
- `name($s)` returns the *name* of the first item of the nodeset $s in document order, *including* its namespace prefix; if $s is omitted, it is taken to be the context item

3.3 String Functions

The following functions apply to character strings.

- `concat($s`$_1$`,...,$s`$_n$`)` *concatenates* the strings `$s`$_1, \ldots,$ `$s`$_n$
- `starts-with($a,$b)` returns `true()` if the string `$a` *starts with* the string `$b`
- `contains($a,$b)` returns `true()` if the string `$a` *contains* the string `$b`
- `substring-before($a,$b)` returns the *substring* of `$a` *before* the first occurrence of `$b`
- `substring-after($a,$b)` returns the *substring* of `$a` *after* the first occurrence of `$b`
- `substring($a,$n,$l)` returns the *substring* of `$a` of length `$l` starting at index `$n` (indexes start from 1). `$l` may be omitted.
- `string-length($a)` returns the *length* of the string `$a`
- `normalize-space($a)` *removes* all leading and trailing *whitespace* from `$a`, and *collapse* all whitespace sequence to a single character
- `translate($a,$b,$c)` returns the string `$a`, where all occurrences of a character from `$b` has been *replaced* by the character at the same place in $c.

3.4 Boolean and Number Functions

- `not($b)` returns the *logical negation* of the boolean `$b`
- `sum($s)` returns the *sum* of the values of the nodes in the nodeset `$s`
- `floor($n)` rounds the number `$n` to the *next lowest* integer
- `ceiling($n)` rounds the number `$n` to the *next greatest* integer
- `round($n)` rounds the number `$n` to the *closest* integer

4 XPath examples

We conclude this introduction with several examples of XPath expressions, in order to get a flavor of the concrete possibilities of the language.

- `child::A/descendant::B`
  `B` elements, descendant of a `A` element, itself child of the context node; It be abbreviated to `A//B`.
- `child::*/child::B`
  All the `B` grand-children of the context node.
- `descendant-or-self::B`
  Elements `B` descendants of the context node, *plus* the context node itself if its name is `B`.
- `child::B[position()=last()]` The last child named `B` of the context node. It abbreviates to `B[1]`.
- `following-sibling::B[1]` The first sibling of `B` (in the document order).
- `/descendant::B[10]` The tenth element of type `B` in the document. But *not* the tenth element of the document, if its type is `B` (recall that axis and node test are evaluated first)!
- `child::B[child::C]` Child elements `B` that have a child element `C`. Can be abbreviated to `B[C]`.
- `/descendant::B[@att1 or @att2]` Elements `B` that have an attribute `att1` or an attribute `att2`. Can be abbreviated to `//B[@att1 or @att2]`.
- `*[self::B or self::C]`]: children elements named `B` or `C`

5 Further readings

There exists a large number of implementations for XPath.

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 .*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.

The W3C published in 2007 a recommendation for XPath 2.0, an extension of XPath 1.0, mostly backward compatible with XPath 1.0. Main differences:

- **Improved data model**, tighly associated with XML Schema. The main novelty is a new *sequence* type, representing ordered set of nodes and/or values, with duplicates allowed. XSD types can also be used for node tests.

- **More powerful** new operators (loops) and better control of the output (limited tree restructuring capabilities)
- **Extensibility** with many new built-in functions; possibility to add user-defined functions.

XPath 2.0 is *also* a subset of XQuery 1.0.
New node tests in XPath 2.0:

item() any node or atomic value

element() any element (eq. to `child::*` in XPath 1.0)

element(author) any element named `author`

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

attribute() any attribute

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, and not in XPath 1.0.

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