Contents

1. XML Path Language (XPath)

A common task for many applications based on XML is to identify certain parts of an XML document. Instead of having each application define its own method for doing this, W3C developed the *XML Path Language (XPath)* [7]. XPath currently is being used by XSLT and XML Schema. However, it is open to be used by other applications as well, and W3C's hope is that XPath will be a common foundation for all applications which need to address parts of XML documents[1]. The benefit of such a widespread usage of XPath would be the re-use of software developed for XPath, and the possibility for human users of XPath-based applications to apply their XPath know-how to new application domains. We therefore consider an understanding of XPath as one of the basic skills for working with XML, and it is worthwhile to spend some time learning it. In addition to this chapter, Kay [11, 13] and Fung [9] provide good resources for understanding and learning XPath.

Depending on the personal preference of learning, this chapter about XPath should be read before or after the XPointer chapter. Even though XPointer builds on XPath, readers with a preference for a top-down approach to learning will find it more suitable for them to first read (or at least skim) the chapter about XPointer, and then continue with XPath as the detailed description of what can be done with XPointers. Readers with a preference for bottom-up approaches, on the other hand, will probably start to learn about XPath's underlying principle of addressing parts of XML documents, before continuing to XPointer's application of this principle for the purpose of defining XML fragment identifiers. Either way, the chapters on XPath and XPointer have a number of interdependencies, making them truly understandable only in combination (at least in terms of supporting linking mechanisms).

The basic idea of XPath (and the reason for its name) is to describe the addressing into an XML document in a sequence of steps, which are specified in a *path notation*. This is intuitive for people who are used to working with hierarchically organized information[2], and can be easily represented in a printable representation, which is one of the key requirements for XPath. While XPath in the context of XSLT will often be hidden inside an XSL style sheet, XPointers using XPath will be visible to users (for example, part of a URI reference, and visible in the address bar of a browser) and should be easily readable and exchangeable by non-electronic means, such as handwriting or even conversation over the phone[3].

XPath can be structured into different areas. The first interesting area to look at is the *general model*, describing which concepts and data types are used in XPath and how the data types can be manipulated. This aspect of XPath is described in Section 1.1. The most widely used construct of XPath is a *location path*, which is explained in detail in Section 1.2. More general than location paths are *expressions*, which are described in Section 1.3. Another important aspect of XPath are *functions*, which can be used in expressions (very similar to a function library in a programming language). They are described in Section 1.4. Finally, to illustrate the concepts described in this chapter, Section 1.5 shows some examples of XPaths and also gives guidelines for constructing XPaths.

[1] W3C currently is working on *XML Query* [4], which will define a data model for XML documents, a set of query operators on that data model, and a query language based on these query operators. If possible, XML Query will also be based on XPath.

[2] In an abstract sense, file systems on computers (which are understood by virtually all computer users), and XML documents are similar, in that they represent hierarchically structured information. This common structure often is referred to as a tree, having the file system's root directory or the XML document element as its root, and then organizing all information starting from that.

[3] In fact, it is one of the key requirements of URIs (and XPath as the foundation of XPointer will be used in URI fragment identifiers) that they can be printed and even exchanged over the phone.

1.1 General Model

In order to understand XPath, it is important to introduce some terms and concepts. Most generally, an XPath is an expression which is evaluated to yield an object. This basic model raises two major questions. Firstly, what are the results of the evaluation of an expression? Second, in which context does this evaluation takes place? To answer these questions we need to consider various concepts that form the foundation of XPath:

- *Object types*
  XPath knows about four object types: a `node-set`; boolean; number; and string values. While the latter three are well-known from other application areas, the concept of the `node-set` is less common. Basically, a `node-set` is nothing more than what is implied by its name, an unordered collection of nodes (which themselves can have different types). It is important to recognize that the set of object types defined by XPath is the minimal set to be supported by all XPath applications. XPath applications (eg, XPointer) are allowed to specify additional object types.
- *Context*
  Each expression is evaluated within a given context. In general, XPath assumes that the context is defined by the application sitting on top of XPath (eg, XPointer or XSLT). A context consists of the following things:
  - A node, which is said to be the *context node*.
  - The *context position* and the *context size* determine the location of the context node in the context, and the overall size of the context. Both values are non-zero positive integers that are used to put the context node into the context of the containing `node-set`.
  - A set of *variable bindings*, which are mappings from variable names to variable values. The values of variables can be of any object type.
  - A *function library*, which is a mapping from function names to functions. Each function accepts zero or more arguments of any object type and returns a single result of any object type.
  - A set of *namespace* declarations in scope for the expression, which consist of a mapping from prefixes to namespace URIs.

  It is important to recognize that the context defined by XPath is the minimal context to be supported by all XPath applications. Because XPath is not intended to be used on its own, actual XPath applications (eg, XPointer) will be based on it, and they are allowed to specify additional context elements, such as new variables and functions.

XPath's object types as well as the context are minimal requirements which can be extended by XPath applications. Furthermore, the context is only defined on a per-expression base, which means that the context changes while evaluating an XPath consisting of multiple expressions. A very simple example for this is an XML document consisting of two levels of hierarchy, and an XPath addressing into one element of the lowest level by first addressing the parent element in the intermediate level, and then, in this new context, addressing the target element. It is this design for stepwise (or hierarchical) addressing that makes XPath very powerful.

The concept of the `node-set` has been introduced already, but so far we have not said exactly what is a node. XPath operates on an abstract tree of nodes, which represents the XML document to which the XPath is applied. (Note that this use of the term *node* is different from the hypertext concept of *node* as a unit of information that should be presented as a whole.) The tree can be seen as derived from the *XML Information Set (XML Infoset)* representation of the document. XPath identifies seven types of nodes, which are explained in detail in the following sections. However, there are some concepts of the data model which are not associated with any particular node type, and these are explained in the following list:

- *String value*
  For every type of node, there is a way to evaluate the `string-value` of that type. For element nodes and root nodes, this is defined by the `nodeValue` method as specified by the *Document Object Model (DOM)*. The `string-value` of a node is important in XPath because it is used in different contexts to perform certain evaluations and comparisons involving nodes.
- *Expanded name*
  Some node types also have an `expanded-name`, which is a representation of a local name (a string), and a namespace URI (empty or a string).
- *Document order*
  All nodes for an document are arranged in a certain order, called the *document order*. This order is determined by the order in which the first character of the XML representation of the node occurs in the XML document. In Figure 1.1, it is shown how the document order would be for elements of an XML document, if they were used in the hierarchical structure depicted in the figure[4]. There also is a *reverse document order*, which is is the reverse of the document order.

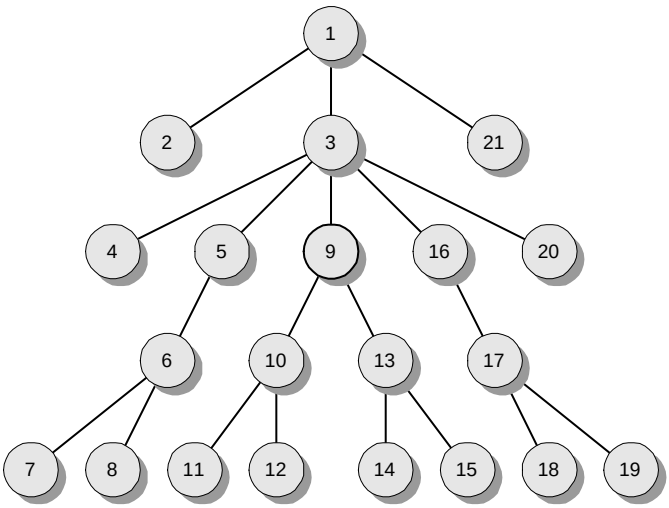

Fig. 1.1 Document order of XPath nodes

  In document order, all nodes are ordered, not only element nodes. Attribute nodes occur after the element node on which the attribute has been used, and before the element nodes of the element's children. Namespace nodes occur before attribute nodes, even if the namespace has been declared after the element's other attributes. The order of attribute and namespace nodes amongst themselves is implementation-dependent.
- *Node relationships*
  The usual terminology for tree-like structures applies to XPath, which means that every node (except the root node) has exactly one *parent*, one node can have any number of *child* nodes, and nodes never share child nodes. Finally, the *descendants* of a node are its child nodes and their descendants.

  With these common concepts for nodes in mind, we will now examine each node type in detail.

1.1.1 Root Node

The root node is the root of the tree and therefore each XML document tree has exactly one root node. The children of the root element are the document element (an element node), and processing instruction

[4] It is important to notice that this figure does not show attribute and namespace nodes.

and comment nodes for all processing instructions and comments occurring before and after the document element. It is very important to keep in mind that the root node is not the node representing the XML document element, which is represented by a child node of the root node.

The root node's `string-value` is the concatenation of the `string-value`s of all text node descendants of the root node in document order.

1.1.2 Element Node

Each element in an XML document is represented by an element node. The children of an element node are the element, comment, processing instruction, and text nodes for the element's content. It is also worth noting that any entity references (both internal and external references as well as character references) are resolved, which means that XPath does not provide any means to access the entity structure of a document.

Every element node has an `expanded-name`, which is evaluated in accordance with the XML Namespaces recommendation. Element nodes may also have a *unique identifier (ID)*, if the element has an attribute with the type `ID` on it[5]. No two element nodes in a document can have the same ID[6].

The `string-value` of an element node is the concatenation of the `string-value`s of all text node descendants of this element node in document order.

1.1.3 Attribute Node

For each attribute of an element, there is an attribute node. Somewhat confusingly, although the element node is the parent of all of its attribute nodes, the attribute nodes are not treated as children of the element node (remember that only element, comment, processing instruction, and text nodes are the children of element nodes). A defaulted attribute is treated as if the attribute has been specified in the document. However, if the default was declared as `#IMPLIED` and the attribute was not defined on the element, then there is no attribute node for the attribute.

Some special attributes (such as `xml:lang` and `xml:space`) by definition have the semantics of implicitly applying to all descendants, unless being overridden. However, this does not mean that all descendants have attribute nodes for these attributes. The attribute nodes for these attributes will only appear at those elements where the attribute was explicitly set in the XML document. Attributes for declaring namespaces (bearing the `xmlns` prefix) will not appear as attribute nodes, but as namespace nodes.

Every attribute node has an `expanded-name`, which is evaluated in accordance with the XML Namespaces recommendation. The `string-value` of an attribute node is its normalized attribute value, with the normalization as defined by the XML recommendation.

1.1.4 Namespace Node

For each namespace in scope for an element, there is a namespace node. As with attribute nodes, the element node is the parent of all of these nodes but the namespace nodes are not children of the element node. The namespace nodes include the namespace of the `xml` prefix (which is defined by the XML recommendation), and the default namespace if one is in scope for the element. The result of this is that namespace nodes will be present for an element for all of the following cases:

- For every namespace declaration on the element (ie, for every attribute whose name starts with the prefix `xmlns`).
- For every namespace declaration on an ancestor of the element, unless the element itself or a nearer ancestor re-declares the namespace.

[5] Here it becomes obvious that XPath requires DTD processing, because the DTD contains the information about attribute types.

[6] If two elements have the same ID (in which case the document is invalid), then the first element in document order is assigned the ID, while the second element does not have an ID.

- For an `xmlns` attribute (the declaration of the default namespace), if the attribute on the element or the nearest ancestor where it occurs is non-empty (using an empty value for the `xmlns` attribute undeclares the default namespace).

Every namespace node has an `expanded-name`, where the local part is the namespace URI that belongs to the namespace, and the namespace URI of the `expanded-name` is always null. The `string-value` of a namespace node is the namespace URI that belongs to the namespace (relative URIs are resolved to absolute URIs).

1.1.5 Processing Instruction Node

For every processing instruction in the document, there is a corresponding processing instruction node (the only exception are processing instructions in the document type declaration). Every processing instruction node has an `expanded-name`, where the local part is the processing instruction's target, and the namespace URI of the `expanded-name` is always null. The `string-value` of a processing instruction node is the part of the processing instruction following the target until the closing "`?>`", including any whitespace.

1.1.6 Comment Node

For every comment in the document, there is a corresponding comment node (the only exception are comments in the document type declaration). Comments nodes do not have an `expanded-name`. The `string-value` of a comment node is the content of the comment, not including the opening "`<!--`" and the closing "`-->`".

1.1.7 Text Node

Character data occurring inside elements is grouped together in text nodes. Each text node holds as much character data as possible, ie all the character data between two tags. Text nodes do not have an `expanded-name`. The `string-value` of a text node is its character data. Text nodes always have at least one character of data.

`CDATA` sections are treated as character data, with every character inside the `CDATA` section resulting in one character in the text node. The `CDATA` markers are not included in the text node. Characters in attribute values, processing instructions, or comments do not produce text nodes.

1.1.8 Example

To illustrate the different node types presented in the previous sections, consider the following simple example XML document:

```
<?xml version="1.0"?>
<?example do not process ?>
<!DOCTYPE People SYSTEM "People.dtd">
<People xmlns="http://www.people.org/NS/People1234">
  <!-- List of people -->
  <Person StaffID="123456">
    Who: <Name>Anna Smith</Name>
    What: <Position>Sales Manager</Position>
  </Person>
  <Person StaffID="987654">
    Who: <Name>Bill Black</Name>
    What: <Position>XML Programmer</Position>
  </Person>
</People>
```

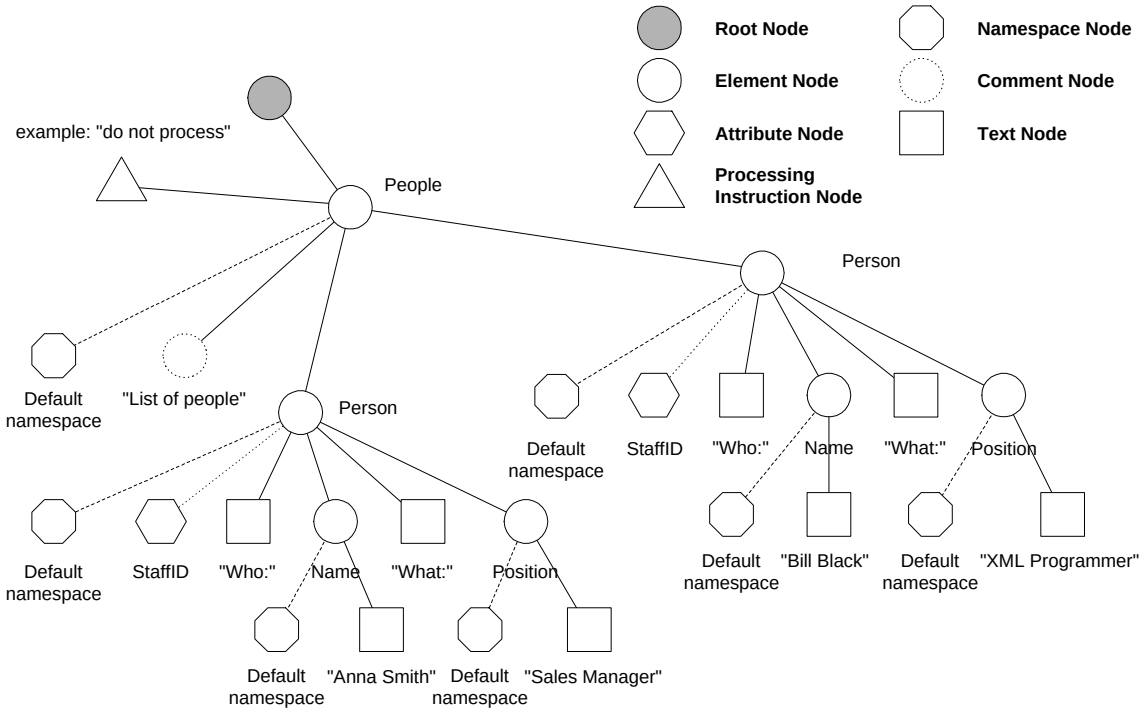

Fig. 1.2 Example XPath node tree

We can represent this XML document as an XPath node tree as shown in Figure 1.2. This figure shows the various XPath nodes. Note, however, that for the sake of clarity, the existence of text nodes for the whitespace in the XML document has been omitted. It is also important to notice that this tree is derived from the concepts introduced in the Infoset, but also has some differences (such as the absence of the document type declaration, and the aggregation of characters into text nodes).

One interesting observation about the node tree is that some nodes do not directly correspond to any XML markup, which is the case for the namespace nodes in the descendant elements of the `People` element, which inherit the default namespace declaration from the `People` ancestor element. Another example of nodes not directly corresponding to any XML markup in the document are defaulted attributes taken from the DTD (this case is not shown in the example).

It is also worth noticing that the XML declaration is not part of the tree (only the other processing instruction is represented as a processing instruction node). Another important thing to note is the kind of relationship between nodes. Solid lines in the tree denote "real" tree-like relationships, where the upper node is the parent of the lower node, and the lower node is a child of the upper node. Dashed lines, on the other hand, denote the special kind of relationship, where the upper node is a parent of the lower node, but the lower node is not a child of the upper node. This kind of relationship within the node tree is used for attribute and namespace nodes.

1.2 Location Paths

Now that we have an understanding of how an XML document is represented by nodes of different types, the next step is to look at how these nodes can be used for addressing into an XML document. The most important construct of XPaths is the *location path*. A location path is used to address a certain `node-set` of a document. This is achieved by concatenating multiple steps into one location path, which describe

with increasing specificy which parts of the document should be addressed. The following definitions are taken from the XPath specification[7] and describe a location path syntactically:

```
 [1] LocationPath          ::= RelativeLocationPath
                             | AbsoluteLocationPath
 [2] AbsoluteLocationPath  ::= '/' RelativeLocationPath?
                             | AbbreviatedAbsoluteLocationPath
 [3] RelativeLocationPath  ::= Step
                             | RelativeLocationPath '/' Step
                             | AbbreviatedRelativeLocationPath
 [4] Step                  ::= AxisSpecifier NodeTest Predicate*
                             | AbbreviatedStep
 [5] AxisSpecifier         ::= AxisName '::'
                             | AbbreviatedAxisSpecifier
 [6] AxisName              ::= 'ancestor' | 'ancestor-or-self'
                             | 'attribute' | 'child' | 'descendant'
                             | 'descendant-or-self' | 'following'
                             | 'following-sibling' | 'namespace'
                             | 'parent' | 'preceding'
                             | 'preceding-sibling' | 'self'
 [7] NodeTest              ::= NameTest
                             | NodeType '(' ')'
                             | 'processing-instruction' '(' Literal ')'
 [8] Predicate             ::= '[' PredicateExpr ']'
 [9] PredicateExpr         ::= Expr
[10] AbbreviatedAbsoluteLocationPath ::= '//' RelativeLocationPath
[11] AbbreviatedRelativeLocationPath ::= RelativeLocationPath '//' Step
[12] AbbreviatedStep       ::= '.' | '..'
[13] AbbreviatedAxisSpecifier ::= '@'?
```

The syntax of location paths has been designed to be similar to other hierarchical notations used in computer applications, such as URIs or file names. A location path is either absolute or relative, where absolute paths are denoted by a leading slash and a trailing relative location path. A relative location path is divided into several steps, separated by slashes. These location steps are described in Section 1.2.1. XPath also defines a number of abbreviations for the most commonly used location paths and steps, and these abbreviations will be mentioned where appropriate.

Before we go into the details of location steps and what can be done by using and combining them, we give some examples of location paths. XPath supports an abbreviated syntax for location paths (as specified in rules `[10]` to `[13]`), which is often used in real-world applications of XPath. We explain these abbreviation mechanisms in detail in Section 1.2.5. However, in the following examples we use the full syntax, which is more verbose and therefore easier to explain and understand:

1. `attribute::name`
   Selects the `name` attribute of the context node.
2. `/descendant::numlist/child::item`
   Selects all the `item` elements that have a `numlist` parent and that are in the same document as the context node.
3. `child::para[position()=1]`
   Selects the first `para` child of the context node.
4. `/descendant::figure[position()=42]`
   Selects the forty-second `figure` element in the document.
5. `/child::doc/child::chap[position()=5]/child::sect[position()=2]`
   Selects the second `sect` child of the fifth `chap` child of the `doc` document element.

[7] We only list XPath grammar productions where they help to understand the concepts behind them. The numbering of the productions has been taken from the XPath specification [7], which should be consulted for a complete and authoritative definition of the XPath grammar. It can be found at http://www.w3.org/TR/xpath.

Table 1.1 Overview of XPath axes

Axis name	Direction	Principal Node Type	Page	Figure
`ancestor`	reverse	`element`	9	1.3 (p. 10)
`ancestor-or-self`	reverse	`element`	9	1.3 (p. 10)
`attribute`	n/a	`attribute`	9	
`child`	forward	`element`	10	1.3 (p. 10)
`descendant`	forward	`element`	11	1.3 (p. 10)
`descendant-or-self`	forward	`element`	11	1.4 (p. 12)
`following`	forward	`element`	11	1.4 (p. 12)
`following-sibling`	forward	`element`	12	1.4 (p. 12)
`namespace`	n/a	`namespace`	12	
`parent`	forward	`element`	13	1.4 (p. 12)
`preceding`	reverse	`element`	13	1.5 (p. 14)
`preceding-sibling`	reverse	`element`	13	1.5 (p. 14)
`self`	forward	`element`	14	1.5 (p. 14)

6. `child::para[attribute::type='warning'][position()=5]`
   Selects the fifth `para` child of the context node that has a `type` attribute with value `warning`.
7. `child::para[position()=5][attribute::type="warning"]`
   Selects the fifth `para` child of the context node if that child has a `type` attribute with value `warning`. If there is no such attribute, nothing is selected.

In all these examples, it has become apparent that two things are very important in locations paths, which are the context (determining in relation to which position in a document a location path is evaluated), and the difference between relative and absolute location paths (easily identified by beginning either with an *axis specifier* or a slash character, as defined by rules `[1]` to `[3]`).

1.2.1 Location Steps

A location step is the most important construct of a location path in XPath, making it possible to select a number of nodes from a given set of nodes according to certain criteria (eg, selecting only the elements of a `node-set` which have a given name or a given relation to the context node). As defined by rule `[3]`, location steps are separated by slash characters. Each location step is defined as consisting of three distinct parts, an axis, a node test, and a predicate. These parts, which are the core building block of every location step (and therefore every location path as well) as described in Sections 1.2.2, 1.2.3 and 1.2.4. To make location steps more compact, XPath also defines a number of abbreviations, which are discussed in Section 1.2.5. Finally, Section 1.2.6 gives examples for XPath location paths and also mentions some of the fine points of using them.

1.2.2 Axes

An axis in XPath defines which nodes are seen starting from the context node. For example, the most often used axis is the `child` axis, and seen from a context node, all child elements of the context node are placed on this axis. Generally speaking, axes can be most easily remembered as a special kind of view from the context node, each defining another particular way to see the nodes of an XML document.

Table 1.1 lists all XPath axes. Apart from additional information to easily locate the axes' description and visualization, it also contains two additional properties of axes:

- *Direction*
  The *direction* of an axis determines in which order the nodes on an axis are arranged. If the axis is a forward axis, then the nodes are arranged in document order, if it is a reverse axis, then they are

arranged in reverse document order (more about document order can be found in Section 1.1). The direction of an axis is very important when nodes on an axis are selected using their position, which is discussed in detail in Section 1.2.4 dealing with location step predicates.

- *Principal node type*
  The *principal node type* of an axis determines the type of nodes being selected by the node test "`*`" (more about node tests in Section 1.2.3), so depending on the node test, it may be important to know the principal node type. However, XPath's rule is that "if an axis can contain elements, then the principal node type is element, otherwise, it is the type of the nodes that the axis can contain", so the principal node type can be easily remembered.

Because axes are most easily remembered as views from the context node, Figures 1.3 to 1.5 visualize the axes, taking the emphasized node in the middle of the tree as context node and shading the nodes which are part of the individual axes. It should be noted, however, that even though in these examples all axes are non-empty, it is perfectly legal for axes to be empty, one simple example being the `child` axis of an element that does not have any child elements. Furthermore, the figures only show element nodes which is the reason why the `attribute` and the `namespace` axes are not shown[8].

Another important thing to remember is that in XPath's node tree, the XML document element is not the root node, but a child (the only element child) of the root node. Consequently, when using axes that select nodes before, above, or after the context node, the root node as well as children of the root node other than the document element node (ie, comment nodes or processing instruction nodes, but not attribute nodes or namespace nodes) may also be part of these axes.

Before we go into the details of all axes available in XPath, we would like to reiterate that according to rule `[4]`, the axis is the first component in every location step (the only exception being an abbreviation or no axis specifier, thereby implicitly specifying the default axis (xpathchild), as defined in rule `[13]`).

- `ancestor` — *Selects all ancestors of the context node*
  This axis selects all ancestor nodes of the context node. The ancestors of the context node are its parent node, the parent node of the parent node and so on until the root node. Consequently, the `ancestor` axis will always include the root node, unless the context node is the root node (in which case the `ancestor` axis will be empty). An easy visualization of the `ancestor` axis is to look up the tree starting from the context node, and all nodes up to the root are on this axis. Because this view of the nodes starts further down in the tree and goes up, the `ancestor` axis is a reverse axis.
- `ancestor-or-self` — *Selects all ancestors including the context node*
  This axis selects all ancestor nodes of the context node and the context node itself. The ancestors of the context node are its parent node, the parent node of the parent node and so on until the root node. Consequently, the `ancestor-or-self` axis will always include the root node. The `ancestor-or-self` axis can be seen as the union of the `ancestor` axis and the `self` axis. An easy visualization of the `ancestor-or-self` axis is to look up the tree starting from the context node, and all nodes up to the root are on this axis, but including the context node. Because this view of the nodes starts further down in the tree and goes up, the `ancestor-or-self` axis is a reverse axis.
- `attribute` — *Selects all attributes of the context node*
  The `attribute` axis selects all attributes of the context node. If the context node is not an element node, or if it is an element node, but the element does not have any attributes, then the `attribute` axis is empty. There are three special things to remember about this axis:

[8] The astute reader will notice that strictly speaking, the node tree shown in the examples is not possible using element nodes only. One reason is that by definition the root node of an XPath node tree is not an element node, the other reason being that even if the root node would be accepted, it would not be legal for the root node to have more than one element child. Therefore, in order to keep the examples valid in the sense of XPath node trees, they can be regarded as the node tree directly under the root node, starting with the document element's node.

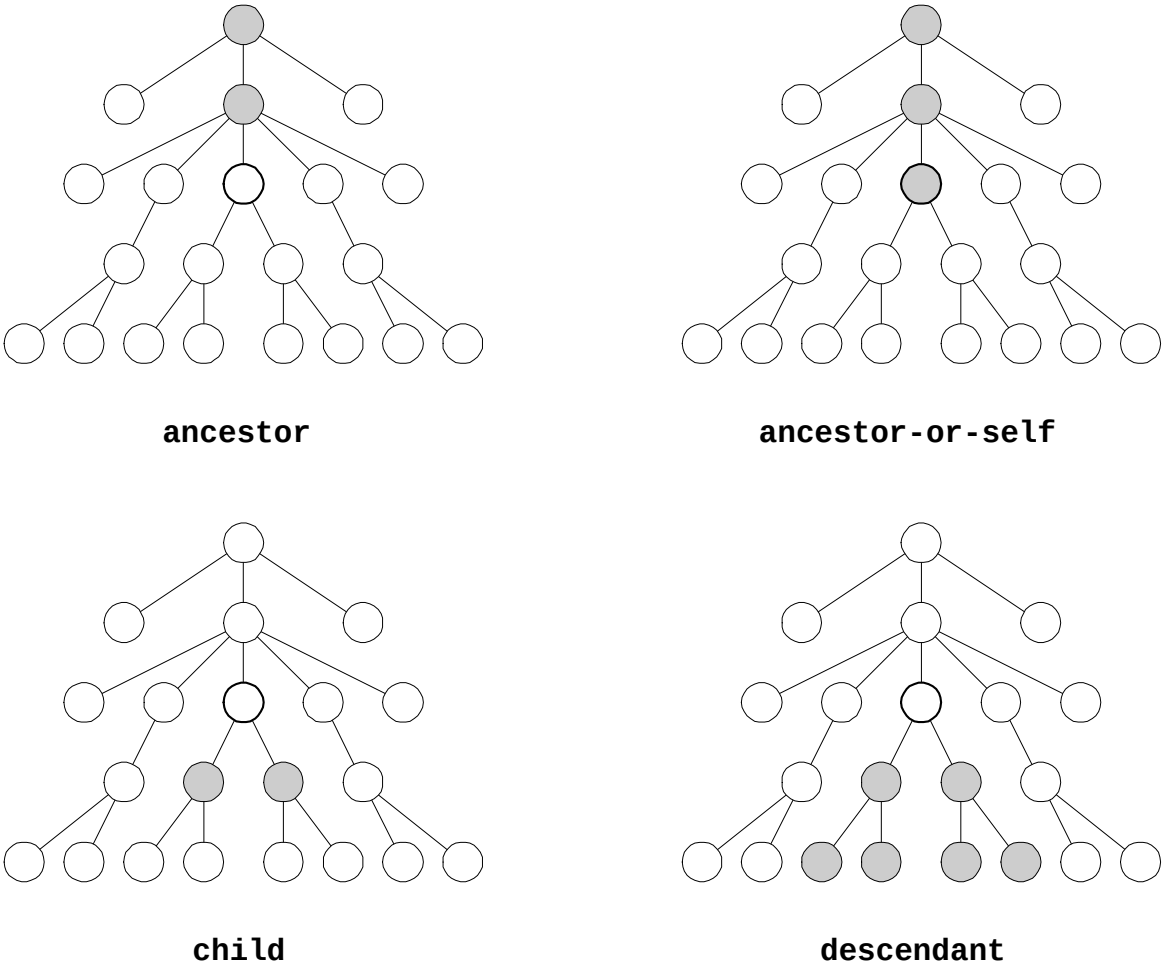

Fig. 1.3 XPath axes `ancestor`, `ancestor-or-self`, `child`, and `descendant`

- Nodes on the `attribute` axis are placed in arbitrary order, so it does not make sense to make any assumptions about the position of attribute nodes on the axis. Therefore, the `attribute` axis does not have a direction.
- The `attribute` axis has a principal node type of attribute nodes, so selecting all nodes on this axis using the "`*`" node test selects attribute nodes.
- Even though namespace declarations syntactically are XML attributes, they do not appear on the `attribute` axis. Instead, all namespace declarations in effect for an element are selected by the `namespace` axis.

The `attribute` axis is one of XPath's most frequently used axes, and it can be conveniently abbreviated using the "`@`" character, as described in detail in Section 1.2.5. Because attribute nodes to not have children, a location step specifying the `attribute` axis usually is the last step in a location path.[9]

- `child` — *Selects all children of the context node*
  The `child` axis is the most frequently used axis of all XPath axes, and for this reason it is the *default axis*. This means that if no axis specifier is given, it is implicitly assumed that the `child` axis should be used (this is formally allowed in rule `[13]`, which allows an empty abbreviated axis specifier). The

[9] However, an attribute node's parent is the element bearing the attribute, so it is possible to further navigate the tree of element nodes starting from an attribute node.

`child` axis selects all children of the context node, which are all nodes located immediately beneath the context node[10]. If the context node does not have any children, then the `child` axis is empty. An easy visualization of the `child` axis is that it selects all nodes which are directly beneath the context node, which in the tree view corresponds to all nodes which are directly connected to the context node.
Even though in most cases the children of the context node will be element nodes (representing the elements directly contained in the element represented by the context node), it is important to remember that the child axis may contain other node types as well, specifically text nodes, comment nodes, and processing instruction nodes. In Section 1.2.3 it will be discussed how these different types of nodes can be differentiated using node tests. If, however, the node test "`*`" is specified, then only nodes of the principal node type will be selected, which in case of the `child` axis are element nodes.
The `child` axis is a forward axis (which can be easily remembered by "looking down" the node tree), so all nodes selected by this axis are arranged in document order.

- `descendant` — *Selects all descendants of the context node*
  The `descendant` axis can be most easily though of as an recursive version of the `child` axis, not only selecting the nodes immediately beneath the context node, but also all nodes which are indirectly beneath the context node (ie, the children's children and so on, until there are no more children)[11]. If the context node does not have any children, then the `descendant` axis is empty. An easy visualization of the `descendant` axis is that it selects all nodes which are directly or indirectly beneath the context node, which in the tree view corresponds to all nodes which are located under the context node.
  As with the child axis, the descendants of the context node will usually be element nodes, but may also be text nodes, comment nodes, and processing instruction nodes. In Section 1.2.3 it will be discussed how these different types of nodes can be differentiated using node tests. If, however, the node test "`*`" is specified, then only nodes of the principal node type will be selected, which in case of the `descendant` axis are element nodes.
  The `descendant` axis is a forward axis (which can be easily remembered by "looking down" the node tree), so all nodes selected by this axis are arranged in document order.
- `descendant-or-self` — *Selects all descendants including the context node*
  The `descendant-or-self` axis is an extended version of the `descendant` axis, selecting all nodes selected by the `descendant` axis, and the context node itself. If the context node does not have any children, then the `descendant-or-self` axis selects only the context node. An easy visualization of the `descendant` axis is that it selects the context node and all nodes which are directly or indirectly beneath the context node, which in the tree view corresponds to all nodes which are located under the context node.
  The `descendant-or-self` axis is a forward axis (which can be easily remembered by "looking down" the node tree), so all nodes selected by this axis are arranged in document order.
- `following` — *Selects all following nodes (in document order)*
  This axis selects the nodes that follow the context node. This axis is rarely used, because it is very specific to the order of elements in the document, and usually XPaths are more likely to be based on structural criteria rather than sequence. The axis can be most easily remembered when thinking of the document in its XML serialization (in contrast to its tree representation), with all nodes being selected whose start tags occur after the end tag of the context element (however, attribute and namespace nodes are never selected by the `following` axis). Therefore, the `following` axis is empty for the last node of the document.
  The `following` axis is a forward axis (which can be easily remembered by selecting the nodes following the context node in the XML serialization), so all nodes selected by this axis are arranged in document order.

[10] It is important to notice that formally, attribute and namespace nodes are not children of element nodes, so these node types are not selected by the `child` axis.
[11] It is important to notice that formally, attribute and namespace nodes are not children of element nodes, so these node types are not selected by the `descendant` axis.

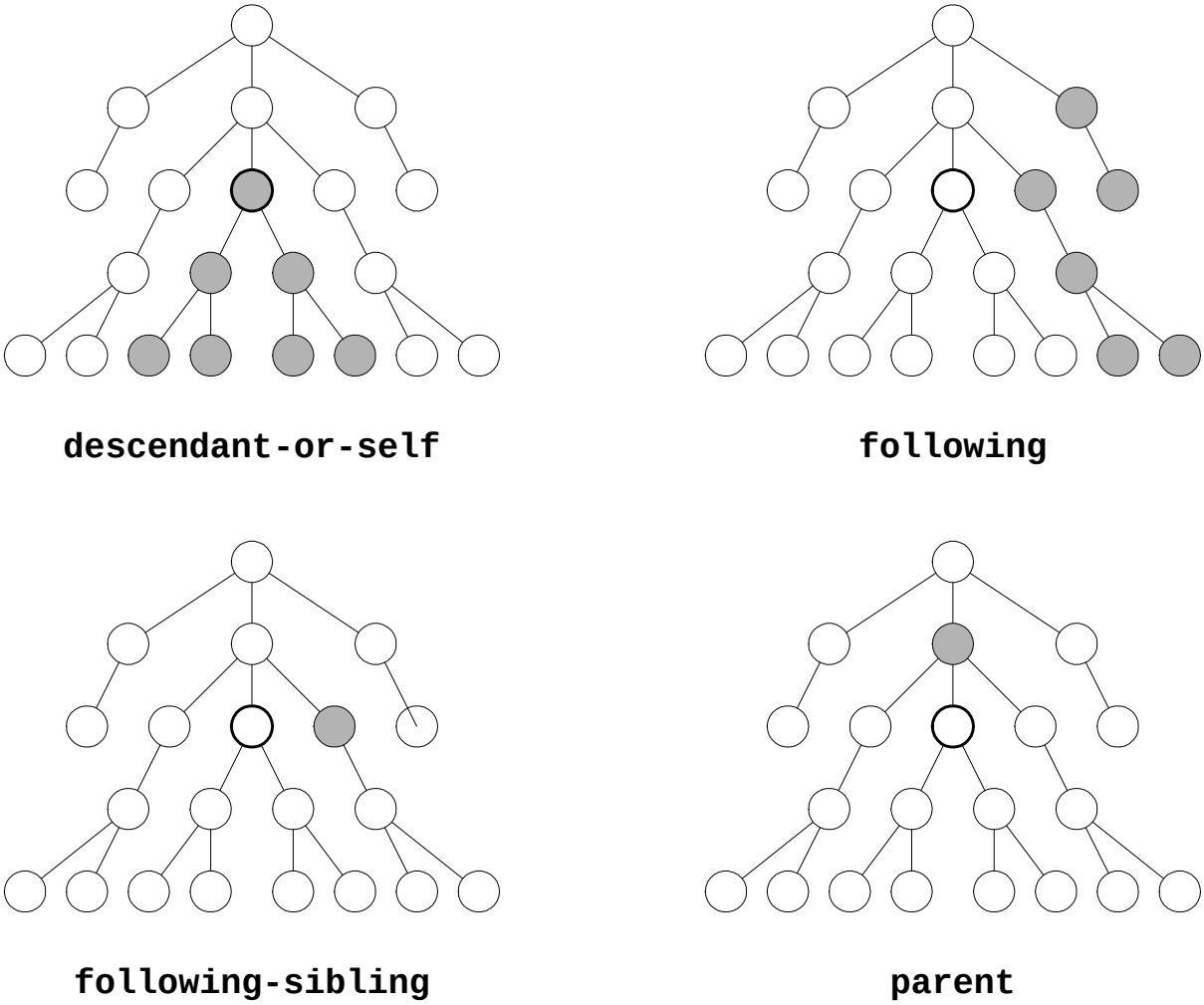

Fig. 1.4 XPath axes `descendant-or-self`, `following`, `following-sibling`, and `parent`

- `following-sibling` — *Selects all following sibling nodes*
  The `following-sibling` axis selects all nodes having the same parent as the context node and occurring after it in document order. This makes it easily possible to select the "next" element when thinking of hierarchy levels in the document tree and when it does not matter how many sub-elements an element may have. The easiest visualization of the `following-sibling` axis is to look from the context node in horizontal direction in document order, and then to only select the nodes which share a common parent with the context node.
  It should be noted that siblings are by definition elements having the same parent element as the context node (otherwise the axis would have been called "cousins of various degrees"...), so the `following-sibling` axis does not select all elements on the same hierarchy level of the XML, but only the elements with the same parent element as the context node. Consequently, if the node is the last child of its parent, then the `following-sibling` axis is empty.
  The `following-sibling` axis is a forward axis (which can be easily remembered by looking in direction of the document order of the node tree), so all nodes selected by this axis are arranged in document order.
- `namespace` — *Selects all namespace nodes of the context node*
  The `namespace` axis selects all namespaces in effect for the context node. If the context node is not an element node, or if it is an element node, but there is no namespace in effect for that element,

then the `namespace` axis is empty. In order to appear on the `namespace` axis, it is not necessary for a namespace to be explicitly defined for that element, because namespace declarations are inherited by child elements. There are three special things to remember about the `namespace` axis:

- Nodes on the `namespace` axis are placed in arbitrary order, so it does not make sense to make any assumptions about the position of namespace nodes on the axis. Therefore, the `namespace` axis does not have a direction.
- The `namespace` axis has a principal node type of namespace nodes, so selecting all nodes on this axis using the "`*`" node test selects namespace nodes.
- Even though namespace declarations syntactically are XML attributes, they do not appear on the `attribute` axis. Instead, all namespace declarations in effect for an element are selected by the `namespace` axis.

Because namespace nodes to not have children, a location step specifying the `namespace` axis usually is the last step in a location path.[12]

- `parent` — *Selects the parent node of the context node*
  This axis selects the parent node of the context node. If there is no parent node (because the context node is the root node), then this axis is empty. Because by definition each node in a tree has at most one parent node, the `parent` axis never selects more than one node. As a convenient abbreviation (and very intuitive for people used to working with file systems), the location step "`..`" can be used to select the parent node of the context node (it is defined in rule `[12]`, more about abbreviations in Section 1.2.5). Because the `parent` axis never selects more than one node, its direction (technically being forward) is irrelevant.
- `preceding` — *Selects all preceding nodes (in document order)*
  This axis selects the nodes that precede the context node. This axis is rarely used, because it is very specific to the order of elements in the document, and usually XPaths are more likely to be based on structural criteria rather than sequence. The axis can be most easily remembered when thinking of the document in its XML serialization (in contrast to its tree representation), with all nodes being selected whose end tags occur before the start tag of the context element (however, attribute and namespace nodes are never selected by the `following` axis). Therefore, the `following` axis is empty for the first node of the document.
  The `preceding` axis is a reverse axis (which can be easily remembered by selecting the nodes preceding the context node in the XML serialization), so all nodes selected by this axis are arranged in reverse document order.
- `preceding-sibling` — *Selects all preceding sibling nodes*
  The `preceding-sibling` axis selects all nodes having the same parent as the context node and occurring before it in document order. This makes it easily possible to select the "previous" element when thinking of hierarchy levels in the document tree and when it does not matter how many sub-elements an element may have. The easiest visualization of the `preceding-sibling` axis is to look from the context node in horizontal direction in reverse document order, and then to only select the nodes which share a common parent with the context node.
  It should be noted that siblings are by definition elements having the same parent element than the context node (otherwise the axis would have been called "cousins of various degrees"...), so the `preceding-sibling` axis does not select all elements on the same hierarchy level of the XML, but only the elements with the same parent element than the context node. Consequently, if the node is the first child of its parent, then the `preceding-sibling` axis is empty.
  The `preceding-sibling` axis is a reverse axis (which can be easily remembered by looking in direction of the reverse document order of the node tree), so all nodes selected by this axis are arranged in reverse document order.

[12] However, an attribute node's parent is the element bearing the attribute, so it is possible to further navigate the tree of element nodes starting from an attribute node.

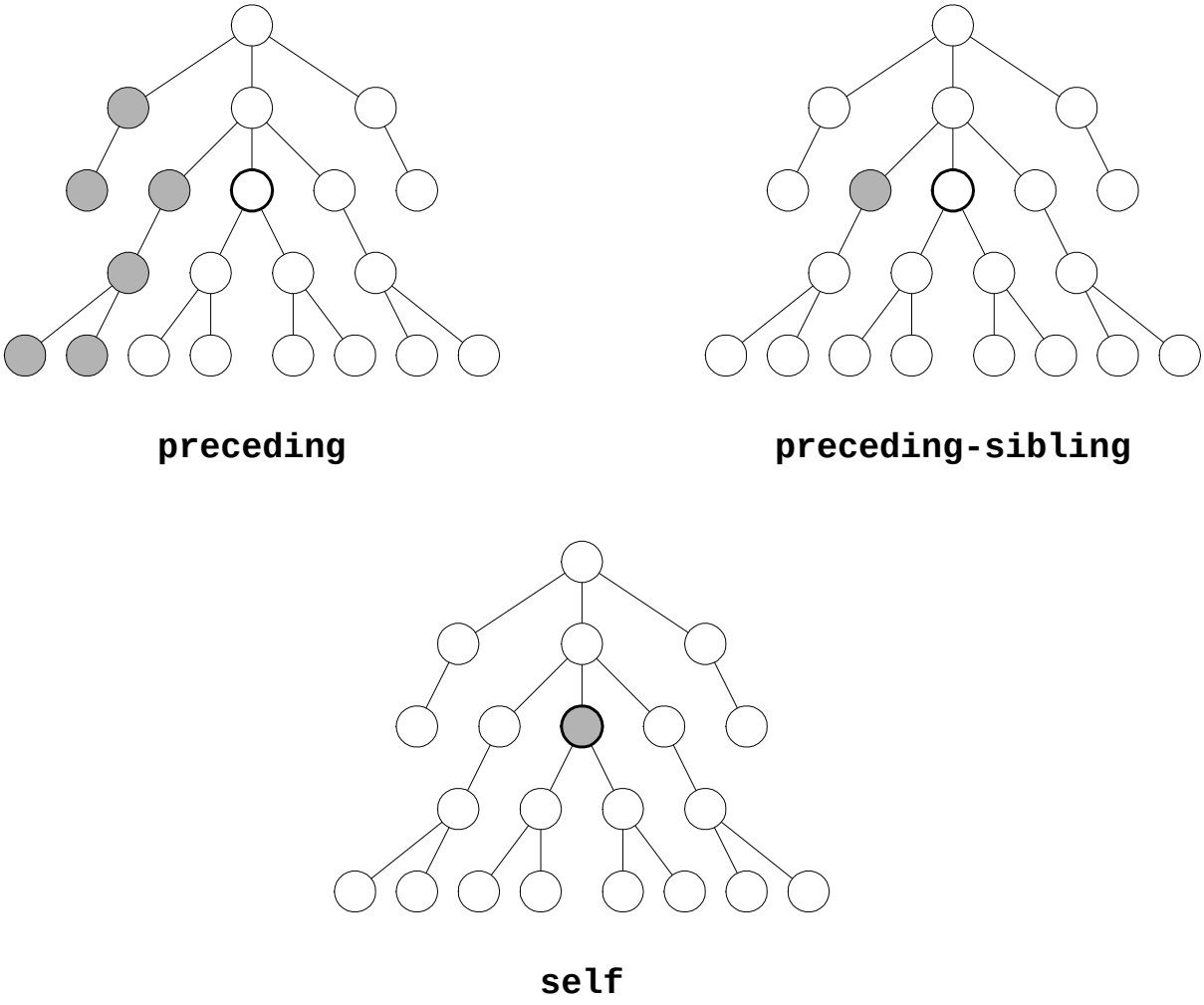

Fig. 1.5 XPath axes `preceding`, `preceding-sibling`, and `self`

- `self` — *Selects the context node itself*
  The `self` axis selects the context node itself. As a convenient abbreviation (and very intuitive for people used to working with file systems), the location step "`.`" can be used to select the context node itself (it is defined in rule `[12]`, more about abbreviations in Section 1.2.5). Because the `self` axis always selects at most one node, its direction (technically being forward) is irrelevant.

While this list of axes may look complex at first, it is the key to creating effective and concise location paths. Of these 13 axes, the `child`, `attribute`, `descendant-or-self`, and `self` axes are used most frequently, often by using their abbreviations as described in detail in Section 1.2.5.

It is also worth noting that the `ancestor`, `preceding`, `self`, `descendant`, and `following` axes partition the document into five disjoint node sets. They do not overlap, and together they select all nodes of the document (excluding attribute and namespace nodes).

1.2.3 Node Tests

Rule `[4]` specifies that a node test is the second element of each location step. Once a node set has been selected using a particular axis, a node test is applied to all these nodes, which potentially reduces the number of nodes in the node set. Looking at XPath's syntax rules for location paths, the following rules are the most important ones for the node test:

```
 [7] NodeTest              ::= NameTest
                             | NodeType '(' ')'
                             | 'processing-instruction' '(' Literal ')'
[37] NameTest              ::= '*'
                             | NCName ':' '*'
                             | QName
[38] NodeType              ::= 'comment'
                             | 'text'
                             | 'processing-instruction'
                             | 'node'
```

Rule [7] shows that a node test either tests for a particular node name, or for a node type (the third case is a special case where only processing instructions nodes of a certain name are selected). If a name test is specified, then only nodes of the principal node type are considered, and the name test may select

- all nodes of the principal node type using the "*" notation,
- all nodes of the principal node and belonging to a certain namespace[13], or
- all nodes of a certain name (which, according to the QName definition from the XML Namespaces recommendation, may or may not specify a namespace prefix).

The most frequent usage of a node test is a name test, testing for nodes of a certain name. However, nodes can also be tested for types, and rule [7] shows that this case is indicated by using parentheses following the type[14]. Because a name test only selects nodes of the principal node type (as shown in Table 1.1), the node() node test is the only node test that selects nodes of more than one type, all other node tests select exactly one type.

Since most of the structural information of an XML document often is identified by element or attribute types, in most XPaths name tests are used which specify a location step for a certain name. This is also apparent through the available abbreviations (described in detail in section 1.2.5), which make it possible to simply use an element's name for specifying a location step among the child axis, and to use the "@" abbreviation for specifying name tests for certain attribute types.

1.2.4 Predicates

According to rule [4], the last component of a location step is an arbitrary number of predicates, though in most cases a location step does not specify any predicate. However, predicates can be used to specify very elaborate filtering criteria, and as such are important for composing complex XPaths. Essentially, a predicate is nothing more than an expression, which is the most general XPath construct. In particular, predicates themselves can be complete XPaths, which are then evaluated using the current context as node set. Predicates are used for further filtering the nodes selected by the axis and the node test (and possibly other predicates), and they are applied to each node in the node set. If a predicate evaluates to true, then the node remains in the resulting node set, otherwise it is removed from the node set. This process is repeated for all predicates of a location step, and the resulting node set of the last predicate is the resulting node set of the whole location step.

In order to completely understand predicates, it is necessary to learn more about XPath's expressions and functions, and these are discussed in Sections 1.3 and 1.4. However, for a first impression of the usage and power of predicates, we give some simple examples in the following XPaths:

- /descendant::chapter[attribute::author][attribute::date]
  This XPath selects all chap elements within the document, and then applies two predicates which themselves contain location paths. In this case, the first predicate filters all chap elements by testing them for an author attribute. The second predicate filters all chap elements that have an author

[13] If a namespace is specified, it is specified using its prefix, and the prefix must be specified somewhere.
[14] Otherwise it would be impossible to syntactically distinguish the name test for elements of the text element type from the text() keyword testing for text nodes.

Table 1.2 XPath abbreviations

Abbreviation	Full XPath Syntax
(no axis specifier)	`child::`
`@`	`attribute::`
`.`	`self::node()`
`..`	`parent::node()`
`//`	`/descendant-or-self::node()/`
`[`x`]`[16]	`[position()=`x`]`

attribute by testing them for a `date` attribute. As a result, this XPath selects all all `chap` elements within the document that have an `author` attribute and a `date` attribute.

- `/descendant::chapter[descendant::figure][descendant::table]`
  Further complicating the example from above, this XPath selects all `chap` elements within the document that have `figure` as well as `table` descendants. Consequently, it selects all chapters that contain figures and tables.

Using location paths inside location step predicates is a very powerful way of selecting nodes, because each predicate is individually evaluated for each node in the node set that goes into the predicate. Constructing this kind of XPaths can take a bit of time, but it can also save a lot of programming (in particular if XPath is used in the context of XSLT), and it certainly is more robust and declarative than a program containing several XPaths and combining their results programatically.

More formally speaking, a predicate filters a node set with respect to the location step's axis to produce a new node set. Taking XPath's general model as described in Section 1.1, for each node in the node set to be filtered, the predicate is evaluated with that node as the context node, with the number of nodes in the node set as the context size, and with the *proximity position* of the node in the node set with respect to the axis as the context position.

The proximity position of a member of a node set with respect to an axis is defined to be the position of the node in the node set ordered in document order if the axis is a forward axis, and ordered in reverse document order if the axis is a reverse axis. It is therefore important to know an axis' direction as shown in Table 1.1.

If the predicate evaluates to `true` for that node[15], the node is included in the new node-set, otherwise, it is not included. This formal definition again refers to XPath expressions, and we therefore discuss predicates in more detail in Section 1.3 about XPath expressions.

1.2.5 Abbreviations

Because one of the design goals of XPath is to provide a concise notation for selecting nodes from an XML document, and because locations paths are the most frequently used form of XPaths (in XSLT style sheets as well as in XPointers), XPath defines some abbreviations for the most frequently used location path components, which are shown in Table 1.2.

These abbreviations cover only a small portion of XPath's features, but they cover many of the most frequently used constructs. The abbreviations provide a very useful mechanism not only making XPaths

[15] If the result of the predicate is not a boolean value itself, then the result will be converted as if by a call to the `boolean` function (more about expressions and functions in Sections 1.3 and 1.4). However, if it is a number, the result will be converted to `true` if the number is equal to the context position, and will be converted to `false` otherwise. This definition can be exploited in several ways, the most popular being the "abbreviation" presented in Table 1.2.

[16] The x in this case is representing any expression evaluating to a number. Technically, this is not an abbreviation, because of the rule that if the result of a predicate is a number, it will be converted to `true` if the number is equal to the context position, and will be converted to `false` otherwise.

shorter, but also helping to make them more easily readable. With the help of the mechanisms, the examples shown on Page 7 can be abbreviated as follows:

1. `attribute::name` → `@name`
   Selects the `name` attribute of the context node. In this case (and this is a very frequently used construct), the attribute axis abbreviation helps to make the XPath more readable.
2. `/descendant::numlist/child::item` → `//numlist/item`
   Selects all the `item` elements that have a `numlist` parent and that are in the same document as the context node. Interestingly, this abbreviation effectively replaces the two-step unabbreviated location path with a three-step abbreviation. However, because the `descendant` step with a name node test can be replaced by two steps (the "`//`" abbreviation meaning `/descendant-or-self::node()/`, and an implicit `child` axis specifying the name[17]) without changing the meaning of the location path, the abbreviated form is preferable because of its conciseness.
3. `child::para[position()=1]` → `para[1]`
   Selects the first `para` child of the context node. As mentioned above, the predicate not really uses an abbreviation, but exploits the mechanism of how predicates are evaluated if the result of the predicate expression is a number.
4. `/descendant::figure[position()=42]` → `/descendant::figure[42]`
   Selects the forty-second `figure` element in the document. In this case, the axis can not be abbreviated because there is no abbreviation for the `descendant` axis. However, the predicate can be specified using the well-known rule for predicates resulting in numbers.[18]
5. `/child::doc/child::chap[position()=5]/child::sect[position()=2]` → `/doc/chap[5]/sect[2]`
   Selects the second `sect` child of the fifth `chap` child of the `doc` document element. This XPath exclusively uses the `child` axis and predicates specifying the position of the children, and it can be seen that in this case, the abbreviation mechanism help to make the XPath much more concise.
6. `child::para[attribute::type='warning'][position()=5]` → `para[@type='warning'][5]`
   Selects the fifth `para` child of the context node that has a `type` attribute with value `warning`. Using the `child` and `attribute` axes, all of the XPath's components can be abbreviated.
7. `child::para[position()=5][attribute::type="warning"]` → `para[5][@type="warning"]`
   Selects the fifth `para` child of the context node if that child has a `type` attribute with value `warning`. If there is no such attribute, nothing is selected. In the same way as in the previous example, XPath's abbreviation mechanisms help to make the XPath much shorter.

While these examples only show a few cases of how XPaths can be abbreviated, they should be sufficient to demonstrate that the abbreviation mechanisms not only make XPaths shorter, but also (and more importantly) more readable. It is therefore advisable to use these mechanisms, and because there are so few of them, getting used to writing abbreviated XPaths is quite easy.

1.2.6 Examples

While XPath provides endless ways to select nodes from an XML document, in the following examples we want to show some general techniques which provide useful tips for constructing location paths (more general examples not being restricted to locations paths can be found in Section 1.5).

- `//@id/..`
  This XPath selects all elements that bear an `id` attribute. It is somewhat computationally expensive because it starts with a "`//`" location step, but this can not be avoided when the whole document

[17] This may sound surprising. However, when using the tree representation of XPath's axes as shown in Figures 1.3 and 1.4, it can be easily seen that these two constructs indeed are identical with respect to their result.

[18] A precautionary note: The location path "`//figure[42]`" does not mean the same as the location path "`/descendant::figure[42]`". The latter selects the 42nd `figure` element counting from the root node, while the former selects all `figure` elements that are the 42nd `figure` children of their parents.

has to be searched for attributes of a certain name. It is worth noting the last location step, which is used to actually select the elements after selecting the `id` attributes.
As an alternative, the XPath "`//*[@id]`" could be used, which yields exactly the same results as the first variant. In the second case, the existence of `id` attributes is tested for in a predicate and not using the attribute axis, as in the first case.

- `//comment()`
  Using this XPath, all comments in a document can be selected, making it is easy to check a document for any comments.
- `//processing-instruction('xml-stylesheet')/..`
  This XPath returns all nodes that contain a processing instruction with the name `xml-stylesheet` (this name is specified in the standard about associating style sheets with XML documents [5]).
- `//a[starts-with(@href,'http://www.w3.org/')]`
  Even though this XPath uses two string functions which are only introduced in Section 1.4.3, it is an interesting example of how predicates can greatly increase the usefulness of XPaths. In this case, and assuming that hyperlinks as defined in (X)HTML are used, all hyperlinks which point to resources on W3C's server are selected by using string functions to further filter `href` attribute values by inspecting whether they start with a certain string.
- `//table//a/ancestor::p[1]`
  Assuming an HTML-like document type (eg, XHTML), this location path can be used to locate all paragraphs that contain hyperlinks (ie, `a` elements) and occur within a table. It will even correctly work for nested tables, because the predicate of the last location step specifies that in case of multiple `p` ancestors[19], only the element which is closest to the `a` element should be selected (in this case it is important to know that the `ancestor` axis is a reverse axis).

These examples show some general techniques for constructing location paths. In particular, in the last example, it becomes obvious that a key point for constructing robust location paths that work in all cases is the knowledge of the document type. Only if the document type is known, it is possible to foresee all possible cases in which a location path has to produce the expected result, and to install safe-guards against special cases (such as the "`[1]`" predicate in the last example, which protects against the rare case of a "`//table//p//table//p//a`" document, which — even though being rather exotic and slightly contrived — would be legal XHTML).

What these examples also show is that location paths are, in themselves, very powerful and the key point of mastering XPath. However they also require additional constructs for further specifying criteria for filtering node sets. Predicates as discussed in Section 1.2.4 are one such case, and we have already used them within our examples. However, the expressions used within predicates are the most general construct of XPath, and they can be used as whole XPaths, not only within predicates. Expressions are therefore the basis of every XPath (a location path, on which we have focused so far, only is a special case of an expression), and we discuss them in detail in the following section.

1.3 Expressions

An expression is the most basic construct of an XPath, and every XPath is an expression (location paths as discussed in Section 1.2 are only special cases of expressions). The formal syntax rules for an expression defined in the XPath standard are too complicated to be of any use for understanding expressions, but basically it can be stated that XPath expressions are recursively defined as being made up of operators and operands, with different types of operators and different operands. To make this abstract definition a little more real, the expression "`2+3`" is made up of two operands (the numbers) and an operator (the plus sign for the additive operator). This XPath expression would evaluate to a number.

[19] One such case would be a paragraph inside a table, with the paragraph indirectly containing another table which in turn contains hyperlinks within paragraphs. This, even though rarely used in practice, would be valid XHTML.

Table 1.3 Overview of XPath operators and their priorities

Operator	Operator name	Priority
`-`	negation	1
`*`	multiplication	2
`div`	floating-point division	2
`mod`	remainder[20]	2
`+`	addition	3
`-`	subtraction	3
`<`	less than	4
`<=`	less or equal than	4
`>`	greater than	4
`>=`	greater or equal than	4
`=`	equal	5
`!=`	not equal	5
`and`	logical and	6
`or`	logical or	7
`|`	union	8

Besides being the most general XPath construct, expressions are particularly important because they appear with predicates as described in Section 1.2.4. Furthermore, even though expressions can be constructed from location paths and operands alone, they often use functions, which are described in detail in Section 1.4. After these general remarks, we now go into the details of XPath expressions.

In general, expressions are made up of operands and operators. As usual in languages for specifying expressions, this pattern can be applied recursively, so that each operand can be an expression. This leads to expressions like "`2+3*5`", which directly leads to the question of operator precedence (ie, if the expression is evaluated from left to right, it would evaluate to `25`, if the usual arithmetic priorities would be applied, it would evaluate to `17`). XPath has a number of operators, and these are assigned priorities, so that the example expression indeed evaluates to `17`. Table 1.3 lists all XPath operators with their priorities, and the rule is that operators with higher priorities (ie, a lower number) are evaluated first, while operators with equal priorities are evaluated left to right.

If the implicit priorities have to be superseded, it is possible to use parentheses to group expressions for forcing a certain evaluation precedence, so "`(2+3)*5`" would result in `25`. Operators are specific for certain operand types, and depending on the type of operator, operands may be converted implicitly to satisfy these requirements (eg, when comparing a string and a number, then the string is converted to a number). These conversions are always performed as if the explicit conversion functions as described in Section 1.4 would have been used. Even though XPath's operator priorities are as expected, for the sake of clarity it is advisable to use parentheses in certain cases, such as when mixing calculations and comparisons, for example "`(2+3)>(2*3)`" (which evaluates to the boolean value `false`).

All operators in Table 1.3 except for the last one operate on one or several of the common object types as described in Section 1.1, which are numbers, string, and booleans. The more unusual object type of XPath is the node set, and while most operators also accept node sets (in particular, the comparison operators), the most interesting operator is the union operator. The union operator is frequently used to join node sets resulting from location paths, for example the XPath "`//ol | //ul | //dl`" evaluates to a node set containing all `ol`, `ul`, and `dl` elements of a document (these are the three types of list elements defined in HTML). Since location paths themselves are nothing but expressions, they can appear as operands within expressions. An even better demonstration for that is the XPath "`//a[ancestor::ul | ancestor::ol]`", which selects all hyperlinks that occur within an `ol` or an `ul` element (an alternative solution to this problem would be the XPath "`//ul//a | //ol//a`", which is probably more expensive to evaluate because it contains several "`//`" location steps).

[20] This operator calculates the remainder from a truncating division according to IEEE 754 [10] (more about XPath numbers and IEEE 754 in Section 1.4.2), and in particular it should be noted that it is not the same as the `%` operator in Java or JavaScript.

Table 1.4 Overview of XPath functions

Function name	Result type	Arguments	Page
boolean	boolean	object	21
ceiling	number	number	22
concat	string	string, string, string*	23
contains	boolean	string, string	23
count	number	node-set	25
false	boolean		21
floor	number	number	22
id	node-set	object	25
lang	boolean	string	21
last	number		25
local-name	string	node-set?	25
name	string	node-set?	26
namespace-uri	string	node-set?	26
normalize-space	string	string?	23
not	boolean	boolean	21
number	number	object?	22
position	number		26
round	number	number	23
starts-with	boolean	string, string	23
string	string	object?	23
string-length	number	string?	24
substring	string	string, number, number?	24
substring-after	string	string, string	24
substring-before	string	string, string	24
sum	number	node-set	23
translate	string	string, string, string	24
true	boolean		21

As with every expression syntax, XPath expressions are very flexible and thus it makes little sense to give a large number of example expressions. However, the examples presented so far should be enough to convince the reader to start playing around with XPath expressions and try to compose powerful XPaths. Combining expressions, functions (to be discussed in the following section), and location paths, Section 1.5 presents some complex examples that demonstrate XPath's versatility and expressiveness.

1.4 Functions

One of the most important components in XPath expressions as discussed in the previous section are XPath's functions. This situation can be compared to programming languages, which also gain a lot of their power and versatility by providing a rich set of functions (through function or class libraries) which can be taken for granted. XPath defines a set of core functions, which are listed in Table 1.4. In this table, each function is listed with its name, the result type, and the arguments. Arguments with a trailing question mark may be omitted, while arguments with a trailing asterisk may occur as often as required (including not at all).

XPath's core functions must be provided by all XPath implementations, so all XPaths only using the core functions are guaranteed to work with any XPath implementation. XPath is intended primarily as a component that can be used by other specifications. Therefore, XPath explicitly mentions that the core function library may be extended by other standards building on top of XPath. In particular, the XPointer standard extends the set of functions.

In the same way, the `document` function, which is very convenient in XSLT style sheets for accessing multiple documents from within one style sheet, is not an XPath core function, but an XSLT extension of XPath. Additionally, XSLT defines a number of other functions which may be used within XPaths in XSLT style sheets. However, instead of listing these functions here, we simply want to make the point that this extensibility of the XPath function library is very useful for extending XPath whenever necessary

in particular XPath applications, but can be confusing for users moving from one XPath application to another (eg, applying their XSLT knowledge to XPointer and then seeing that some of the functions are not supported in this new environment). Consequently, whenever missing a function that has been seen elsewhere in an XPath-based environment, it is probably an extension of XPath and not one of XPath's core functions.

In the following sections, we give detailed explanations of all XPath core functions, grouped by their type (ie, the type of object they primarily are designed for). Since XPath knows four object types (booleans, numbers, strings, and node sets), there are four sections discussing the functions.

1.4.1 Boolean Functions

Boolean functions return a boolean value, which means their result is either `true` or `false`. Because XPath does not have a way of denoting the boolean values themselves, there are two functions which always return the same value. Consequently, if it is necessary to denote a boolean value in an XPath, the `true()` or `false()` functions must be used. Two important boolean "functions" are not listed here, because they are operators rather than functions, and these are the logical `and` and `or` operators as well as all of the comparison operators explained in Section 1.3. These operators are frequently used to calculate boolean values, for example when testing for multiple values as in "`(@author='dret') or (@author='dbl')`". Apart from these operators producing boolean results, XPath defines the following core functions:

- `boolean` — *Conversion to a boolean value*
  Signature: `boolean boolean(object)`
  Conversion to a boolean value can be done with arguments of all possible object types. A `number` is `true` if and only if it is not zero. A `node-set` is `true` if and only if it is non-empty. A `string` is `true` if and only if its length is greater than zero. Any other object type is converted to boolean according to that object type (ie, as defined in the specification introducing that object type).
- `false` — *Always returns false*
  Signature: `boolean false()`
- `lang` — *Testing for languages of nodes*
  Signature: `boolean lang(string)`
  This function is used to test for a specific language of a node. In XML, the language of a node is specified by the `xml:lang` attribute (as defined by the XML recommendation) which specifies the language according to Internet RFC 3066 [1]. If the language of the context node (or the nearest ancestor specifying a language, if the context node does not specify one) is the same or a sub-language of the language specified in the argument, then the `lang` function returns `true` otherwise it returns false.
- `not` — *Inverting a boolean value*
  Signature: `boolean not(boolean)`
  This function inverts a boolean value, returning `false` when the argument is `true`, and returning `true` when the argument is `false`.
- `true` — *Always returns true*
  Signature: `boolean true()`

One important thing to remember is that the `boolean` function often is used implicitly, because location path predicates are always converted to a boolean value (the one exception being a predicate that evaluates to a number, in which case the result is converted to a boolean based on a comparison with the context node for which the predicate is evaluated).

For example, the location step "`chap[.//figure]`" selects all `chap` elements having `figure` descendants. This location step is equivalent to the variant "`chap[boolean(.//figure)]`", which makes explicit the fact that the predicate's value (in this case, a node set) is converted to a boolean value in order to

determine whether a node is part of the location step's resulting node set. Only if the the node set resulting from evaluating "`.//figure`" for each `chap` is not empty, the corresponding node will become a member of the result node set.

1.4.2 Number Functions

XPath relies heavily on IEEE 754 [10], which is a standard for floating point arithmetic. Even though it is a good idea to rely on a standardized model, IEEE 754 includes some concepts which, from a mathematical point of view, make sense, but can take some time getting used to.

The IEEE 754 standard includes not only positive and negative sign-magnitude numbers, but also *positive* and *negative zeros*, *positive* and *negative infinities*, and a special *Not a Number (NaN)* value. The `NaN` value is used to represent the result of certain operations such as dividing zero by zero. Except for `NaN`[21], floating-point values are ordered; arranged from smallest to largest, they are negative infinity, negative finite nonzero values, negative zero, positive zero, positive finite nonzero values, and positive infinity. Positive zero and negative zero compare equal.

For handling numbers according to the rules of IEEE 754, XPath defines the following core functions:

- `ceiling` — *Rounding up a number*
  Signature: `number ceiling(number)`
  Rounding up a number according to the rules specified in IEEE 754 means to return the smallest number that is not less than the argument and that is an integer. In particular, this means that negative numbers are rounded towards zero (`ceiling(-4.5) = -4`).
- `floor` — *Rounding down a number*
  Signature: `number floor(number)`
  Rounding down a number according to the rules specified in IEEE 754 means to return the largest number that is not greater than the argument and that is an integer. In particular, this means that negative numbers are rounded towards negative infinity (`ceiling(-4.5) = -5`).
- `number` — *Converting to a number*
  Signature: `number number(object?)`
  This function is used to convert its argument to a number. Depending on the type of the argument, the function performs this conversion as follows:
  - A `boolean` value of `true` is converted to `1`, a value of `false` is converted to `0`.
  - A `string` is converted to a valid numeric value if it contains whitespace, followed an optional minus sign, a number (digits optionally including a decimal point), and whitespace.[22] If the string does not adhere to this formatting, it is converted to `NaN`.
  - A `node-set` is converted as if the original argument has been given as argument to the `string` function, and the resulting string has been converted by using it as a string argument to the `number` function.

  Any other object (ie, an object being of another type than the basic types defined by XPath) is converted to a number in a way that is dependent on that type and should be specified in the definition of that type. If the argument is omitted, it defaults to a node-set with the context node as its only member.

[21] `NaN` is unordered, so the comparison operators "`<`", "`<=`", "`>`", and "`>=`" return `false` if either or both operands are `NaN`. The equality operator "`=`" returns `false` if either operand is `NaN`, and the inequality operator "`!=`" returns `true` if either operand is `NaN`. In particular, "`x!=x`" is `true` if and only if `x` is `NaN`.

[22] It should be noted that this specification excludes many common number formats using exponential notations or notations including thousands separators from being converted to a number. Improved functionality for dealing with various number formats will be incorporated into future version of XPath (as discussed in Section 1.6).

- `round` — *Rounding to the next closest integer number*
  Signature: `number round(number)`
  This function returns a number that is closest to the argument and that is an integer. For the special cases of IEEE 754 values (`NaN`, positive and negative infinity, positive and negative zero), the function returns the value of its argument. For numbers less than zero but greater than or equal to `-0.5`, negative zero is returned.
- `sum` — *Summing the string-values of all nodes*
  Signature: `number sum(node-set)`

Even though IEEE 754's definitions of floating point arithmetic may be hard to remember at first sight, it should also be remembered that most of the arithmetic with XPath will be integer arithmetic, and as such is not as complicated as it might seem at first sight. Some of the most frequent uses of numbers in XPath are context positions, and these are always positive integers, so arithmetic with context positions is rather simple.

1.4.3 String Functions

String functions are frequently used for inspecting attribute or element contents, and because in many applications allow some sort of free form data as content, it is very useful to have more sophisticated functions than the simple comparisons which may test strings for equality. In particular, the following core functions operating on strings are defined by XPath:

- `concat` — *Concatenates two or more strings*
  Signature: `string concat(string, string, string*)`
  This function returns the concatenation of its arguments. It must have at least two and can have as many arguments as necessary, all of which must be strings.
- `contains` — *Tests for containment of one string in another*
  Signature: `boolean contains(string, string)`
  If the first argument string contains the second argument string, then this function returns `true`, otherwise it returns `false`. Unfortunately, this function does not provide case-insensitive matching, so if this is required by an application, it must be specified on the application level.
- `normalize-space` — *Normalizes whitespace in a string*
  Signature: `string normalize-space(string?)`
  The `normalize-space` function returns the argument string with whitespace normalized by stripping leading and trailing whitespace and replacing sequences of whitespace characters by a single space. Whitespace characters are the same as those defined in XML, which are space characters, carriage returns, line feeds, and tabs. If the argument is omitted, it defaults to the context node converted to a string.
- `starts-with` — *Tests if one string starts with another*
  Signature: `boolean starts-with(string, string)`
  This function tests whether the first argument starts with the second argument. If this is the case, the function returns `true`, otherwise it returns `false`.
- `string` — *Converting to a string*
  Signature: `string string(object?)`
  The `string` function is used to convert its argument to a string. The argument may be of any type, and depending on the argument's type, the function performs this conversion as follows:
  - If the argument is a node set, it is converted by returning the string value of the node in the node set that is first in document order. For an empty node set, an empty string is returned.
  - Numbers are converted to strings in the following way:
    - `NaN` is converted to the string `"NaN"`.

  - Positive and negative zero are converted to the string `"0"`.
  - Positive and negative infinity are converted to the strings `"Infinity"` and `"-Infinity"`, respectively.
  - Integers are converted to a string of the decimal representation of the number with no leading zeros or separators, negative number are preceded by a minus sign.
  - Otherwise, the number is represented as a floating point number in normal notation with no exponential notation.
- The boolean values `true` and `false` are converted to the strings `"true"` and `"false"`, respectively.

Any other object (ie, an object being of another type than the basic types defined by XPath) is converted to a string in a way that is dependent on that type and should be specified in the definition of that type. If the argument is omitted, it defaults to a node-set with the context node as its only member.

- `string-length` — *Number of characters in a string*
  Signature: `number string-length(string?)`
  The `string-length` function returns the number of characters in a given string. If the argument is omitted, it defaults to the string value of the context node.
- `substring` — *Extracts a substring from a string*
  Signature: `string substring(string, number, number?)`
  This function extracts a substring from a string. The first argument is the string itself, and the second argument specifies the position from which the substring should be extracted[23]. The optional third argument specifies the length of the string to be extracted. If the third argument is not present, the function returns the substring starting at the position specified in the second argument and continuing to the end of the string
- `substring-after` — *Selection after a matching string*
  Signature: `string substring-after(string, string)`
  The `substring-after` function returns the substring of the first argument that follows the first occurrence of the second argument. If the second argument does not occur in the first argument, it returns the empty string. As an example, `substring-after("dret@transcluding.com","@")` returns `"transcluding.com"`.
- `substring-before` — *Selection before a matching string*
  Signature: `string substring-before(string, string)`
  This function returns the substring of the first argument that precedes the first occurrence of the second argument. If the second argument does not occur in the first argument, it returns the empty string. As an example, `substring-after("dbl@transcluding.com","@")` returns `"dbl"`.
- `translate` — *Replacing characters in a string*
  Signature: `string translate(string, string, string)`
  The `translate` function is used to translate the string given as the first argument by substituting all occurrences of the characters in the second argument with the corresponding characters in the third argument[24]. If the third argument string is shorter than the second argument string, then the characters of the second argument string which do not have a corresponding character in the third argument string are removed from the first argument string. A standard application of this function is case conversion, other possible applications include substituting or removing special characters within strings, such as in case of `translate("++41-1-6325132","+-","0")` for converting a printable phone number to the dial string `"004116325132"`.

Even though this repertoire of string functions is useful and sufficient for many applications, it is pretty limited when being compared to really powerful string matching mechanisms, such as regular

[23] It is important to notice that counting starts with 1 (which is different from Java, JavaScript, or C conventions), so `substring("123",2)` returns `"23"`.
[24] Unix users will notice that this is very similar to the standard `tr` utility.

expressions [8]. It would have been nice to have state-of-the-art regular expressions in XPath, but the designers chose to concentrate on defining XPath as a language for mainly working on XML structures. If versatile string matching is required by an application, XPath should only be used for extracting the relevant attributes and elements from the XML document, and then a language more appropriate for the task (such as *Perl* [15]) should be employed.

1.4.4 Node Set Functions

The node set is the most interesting object type of XPath, on the one hand because this is the object type returned by a location path, and on the other hand because a node set directly corresponds to parts of the XML document. By far the most useful "function" for processing a node set is a location path, which can be regarded as a number of "functions" (the location steps) chained one after the other, and each passing its results to the next. However, some functions can not be achieved using location steps alone (or should be available in predicates), and in particular this is true for functions returning a result other than a node set (location steps and thus location paths always result in node sets). The following core functions are available for node sets:

- `count` — *Number of nodes*
  Signature: `number count(node-set)`
  This function returns the number of nodes in a node set. As a simple but useful example, you can count the number of hyperlinks on an (X)HTML page by using `count(//a)`.
- `id` — *Node set with elements selected by ID*
  Signature: `node-set id(object)`
  XML elements may be uniquely identified (within the scope of an XML document) with an attribute of the `ID` type[25]. The `id` function can be used to select elements according to this identification according to the following rules:
  - If the argument is a node set, then the result of the `id` function is the union of applying the `id` function to the string value of each of the individual nodes.
  - For other argument types, the argument is converted to a string as if by a call to the `string` function, and the resulting string is then split into a list of tokens separated by whitespace. For each of the tokens, the element having an ID attribute with that value (if present in the document) becomes part of the resulting node set.

  As an example, considering a document giving chapters individual `ID`s via `ID` type attributes, the function `id("references index")` results in a node set containing two elements, if the document contains two elements with these `ID`s.
- `last` — *Numeric pointer to the last set member*
  Signature: `number last()`
  This function returns a number equal to the context size of the context within which the expression is evaluated. As a frequently used application, the XPath `//chap[last()]` returns the last `chap` element of a document.
- `local-name` — *Returns the local part of the first node*
  Signature: `string local-name(node-set?)`
  The `local-name` function returns the local part of the first node (in document order) of the argument's node set. If there is no such name, or the node set is empty, then it returns the empty string. If no argument is specified, then it defaults to a node set with the context node as the only member.

[25] It is important to notice that the attribute providing the unique ID may have any name, but it has to be declared as being of the type `ID` (remember that the type of an attribute is specified in the document's DTD). Consequently, if a document does not have a DTD, then no element in the document will have a unique ID.

- `name` — *Returns the expanded name of the first node*
  Signature: `string name(node-set?)`
  This function returns the qualified name (ie, the namespace URI as well as the local name) of the first node (in document order) of the argument's node set. If there is no such name, or the node set is empty, then it returns the empty string. If no argument is specified, then it defaults to a node set with the context node as the only member. The namespace URI must reflect the namespace declarations in effect for the node for which the function is evaluated.
- `namespace-uri` — *Returns the namespace URI of the first node*
  Signature: `string namespace-uri(node-set?)`
  The `namespace-uri` function returns the namespace URI of the first node (in document order) of the argument's node set. If there is no such URI, or the node set is empty, then it returns the empty string. If no argument is specified, then it defaults to a node set with the context node as the only member.
- `position` — *Numeric pointer to the context position*
  Signature: `number position()`
  The `position` function returns a number equal to the context position of the context node for which the expression is evaluated. As a frequently used application, the XPath `chap[position()=3]`[26] returns the third `chap` child of the context node (or an empty node set if there are less than three `chap` children).

XPath's node set functions are often used for getting access to information about the XML document itself, such as in the XPath "`name(id('intro'))`", which returns the name of the element bearing the ID `intro`. However, the most frequently used node set functions are probably `count` ("`count(//a)`": how many hyperlinks are in the document?), `last` ("`chap[last()]`": select the context node's last chapter child), and `position` ("`chap[position()=3]`": select the context node's third chapter child).

1.5 Examples

At this point it is worthwhile making a few general remarks about XPath. XPath is essentially a query language for XML documents[27], and considering that XML documents may be quite big, some performance questions should be taken into account when using XPath:

- *Be as specific as possible*
  Consider a document type that always contains `chap` elements as children of `part` elements, which in turn are always children of the `doc` document element. An XPath selecting all chapters within this type of document could either be specified as "`//chap`", or as "`/doc/part/chap`". While the results of both XPaths would always be the same, the second XPath could be evaluated much faster, because the XPath implementation would not have to search the entire document tree for `chap` elements, as specified by the first XPath. It is important to notice that this kind of XPath optimization requires some knowledge about the document type, and because XPath implementations do not interpret or even know about the document type[28], they can not make these optimizations.
- *Filter as early as possible*
  Even though location steps do not necessarily decrease the cardinality of the node set, they very often do so. In many cases, the node set gets smaller with (almost) every location step. Consider a document containing thousands of `address` elements, which in turn contain `city` elements with

[26] As a reminder, this can be abbreviated to `chap[3]`, as described in Section 1.2.5.
[27] Strictly speaking, it is not a complete query language, because it does not allow the recombination of different result sets, but its addressing capabilities cover a substantial area of a query language.
[28] It would be possible to think of an XPath implementation that actually interprets the document type and uses this information for automatic optimizations, but to our knowledge no such implementations exist at the time of writing.

a `zip` attribute. Selecting all addresses with a certain zip code could either be done by specifying "`//address/city[@zip='94704']/..`", or by using "`//city[@zip='94704']/parent::address`". While the optimization presented in the previous paragraph required knowledge about the document type, the optimization in this example requires knowledge about the actual document being used. If there is a large number of `address`es, and the `city` element only rarely occurs in locations other than as an `address` child, then the second XPath is preferable because it filters more specifically with its second step (after the first abbreviated "`//`" step) than the first XPath. If, however, the `city` element also frequently occurs in other contexts than as an `address` child, then the first XPath may be better, because it restricts the search among `city` elements to `address` children.

While XPath in the context of XPointer may be less of a performance problem, because the XPointer is only evaluated once to select the fragment of the XML document, XPaths in XSLT style sheet may be evaluated very often, due to loops or recursions in the style sheet. In this case, XSLT's `xsl:key` element and the `key` function can be used to declare keys that can then be handled more efficiently by the XSLT processor.

The questions discussed so far relate to a specific aspect of an XPath — namely how efficiently it can be evaluated[29]. However, there is another aspect of XPaths, and this is the persistence or robustness of them, meaning how well they work for different documents of the same document type, or how robust they are against changes in the document. This is a very typical problem for XPointer, which is used to point into XML documents, and these pointers should remain valid even if the document is modified, at least to a certain extent.

After these general remarks about composing XPaths, a few examples are intended to show how the general principles of XPath (location paths, expressions, and functions) can be used to select parts of XML documents in a very concise way. In the following examples, we assume that an XHTML document type is being used, which makes the XPaths more understandable. Instead of explaining the syntactical details of the XPaths, we give the intention behind them and leave it as an exercise to the reader to figure out the exact way how it is done (and maybe to compose different XPaths that produce the same result).

- `//a[starts-with(@href,'http:') or starts-with(@href,'ftp:')]`
  This XPath selects all hyperlinks that are referencing resources on HTTP or FTP servers.
- `substring-after(substring-before(//ul[@id='biblio']/li[`x`],']'),'[')`
  Assuming a bibliography using an uniquely identified unnumbered list of list items, where each bibliographic entry is labeled with an identifier occurring between brackets (in the same way as in the bibliography for this book), this XPath selects the label of the xth bibliography entry.
- `//table[not(thead)]`
  If a document should be searched for tables that do not specify a table head (which is not required by XHTML, but should be used to clearly mark the head rows of a table), then this XPath can be used to select all these tables.
- `//h2[normalize-space(string())='Erik Wilde']/preceding::h1[1]`
  Assuming that each first-level heading is followed[30] by a second-level heading containing the section author's name, this XPath could be used to select all first-level headings of sections which have been written by the specified author. The `normalize-space` function is used here as a safe-guard against whitespace characters in the `h2` element's content.

These examples only show a couple of possible applications of XPath. Possibly the best way to learn XPath is to take some documents from your own application domain, and try to write down XPaths that select the required parts of these documents, according to a number of usage scenarios.

[29] Even though it could be argued that optimization should be handled by XPath implementations anyway, at least as much as possible.

[30] It is important to notice that in (X)HTML, headings do not contain the section's content, but precede it. This is unlike most other hierarchical document models, where sections of different hierarchy levels are reflected as hierarchy levels in the document tree.

As a concluding remark, it should be remembered that XPaths are sensitive to object types, and failing to correctly specify an object's type (by using conversion functions or correctly delimiting strings) may result in an XPath that is still valid, but behaves not as expected. In particular, strings must always be delimited, otherwise they are interpreted as location paths (if a location path is permitted at this place), which will produce a completely different result (but not an error message from the application interpreting the XPath).

1.6 Future Developments

XPath is stable in its current version (1.0), and a number of implementations are available, most of them integrated into XSLT processors. Since XSLT and its ability to transform XML documents is one of the key components of an XML-based infrastructure, XSLT is evolving pretty rapidly. At the time of writing, *XSLT 2.0* [12] is being developed[31].

During the work on XSLT 1.1, it has become apparent that XSLT lacks some features which can not be added by simply changing XSLT. Consequently, XSLT 2.0 will be based on *XPath 2.0* [2]. The exact outcome of these activities is still unclear, but generally speaking (and citing from the requirements analysis), the following goals should be accomplished with XPath 2.0:

- Simplified manipulation of XML Schema-typed content
- Simplified manipulation of string content
- Support for related XML standards
- Improved ease of use
- Improved interoperability
- Improved internationalization support
- Maintenance of backward compatibility
- Improved processor efficiency

One of the most important observations is that XPath 2.0 should be fully compatible with XPath 1.0, in such a way that each version 1.0 XPath interpreted by a version 2.0 XPath implementation should yield the same result. It is not yet clear whether that goal will be fully reached, but the required changes to the version 1.0 XPaths (if any) will be kept to a minimum.

Being the basis for a number of XML technologies (in particular XSLT, XPointer, XML Schema, and XML Query), XPath receives a lot of attention and also a lot of input from other standardization efforts. The latest development is *XML Query (XQuery)* [3], which is an effort to define a powerful query language for XML. At the time of writing, XQuery still is in its early development stages, but it is very well possible that XQuery's development will result in more changes to XPath 2.0 than currently under consideration (at the time of writing, the changes are mostly derived from requirements for XSLT 2.0).

The best thing we can therefore do is to advise readers to frequently check `http://www.w3.org/TR` for the latest developments of XPath and related standards. However, XPath 1.0 will remain as it is today, and it will also remain the foundation of XPointer for some time to come (at the time of writing, there is no new version of XPointer under development).

[31] Standardization of *XSLT 1.1* [6] has been stopped to concentrate the efforts on XSLT 2.0. However, the draft version of XSLT 1.1 is already implemented by some of the available XSLT processors. XSLT 1.1 still uses XPath 1.0.

References

1. Harald Tveit Alvestrand. Tags for the Identification of Languages. Internet best current practice RFC 3066, January 2001.
2. Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. XML Path Language (XPath) 2.0. World Wide Web Consortium, Working Draft WD-xpath20-20011220, December 2001.
3. Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, Jérôme Siméon, and Mugur Stefanescu. XQuery 1.0: An XML Query Language. World Wide Web Consortium, Working Draft WD-xquery-20011220, December 2001.
4. Don Chamberlin, Peter Fankhauser, Massimo Marchiori, and Jonathan Robie. XML Query Requirements. World Wide Web Consortium, Working Draft WD-xmlquery-req-20010215, February 2001.
5. James Clark. Associating stylesheets with XML documents. World Wide Web Consortium, Recommendation REC-xml-stylesheet-19990629, June 1999.
6. James Clark. XSL Transformations (XSLT) Version 1.1. World Wide Web Consortium, Working Draft WD-xslt11-20010824, August 2001.
7. James Clark and Steven J. DeRose. XML Path Language (XPath) Version 1.0. World Wide Web Consortium, Recommendation REC-xpath-19991116, November 1999.
8. Jeffrey E. F. Friedl. *Mastering Regular Expressions*. O'Reilly & Associates, Sebastopol, California, January 1997.
9. Khun Yee Fung. *XSLT: Working with XML and HTML*. Addison Wesley, Reading, Massachusetts, December 2000.
10. Institute of Electrical and Electronics Engineers. IEEE Standard for Binary Floating-Point Arithmetic. IEEE Std 754-1985, 1985.
11. Michael Kay. *XSLT Programmer's Reference*. Wrox Press, Chicago, Illinois, April 2000.
12. Michael Kay. XSL Transformations (XSLT) Version 2.0. World Wide Web Consortium, Working Draft WD-xslt20-20011220, December 2001.
13. Michael Kay. *XSLT 1.1 Programmer's Reference*. Wrox Press, Chicago, Illinois, March 2001.
14. *Proceedings of XML Europe 2000*, Paris, France, June 2000.
15. Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly & Associates, Sebastopol, California, 3rd edition, July 2000.