Chapter 8

XML

Most research on data integration has, as described in the previous chapters of this book, centered on the relational model. In many ways, the relational model (and the Datalog query language) are the simplest and cleanest formalisms for data and query representation, so many of the fundamental issues were considered first in that setting.

However, as such techniques are adapted to meet real-world needs, they typically get adapted to incorporate XML (or its close cousin JSON[1]). For instance, IBM's Rational Data Architect or Microsoft's BizTalk Mapper both use XML-centric mappings.

The reason for this is straightforward. XML has become the default format for data export from both database and document sources; and many additional tools have been developed to export to XML from legacy sources (e.g., COBOL files, IMS hierarchical databases). Prior to XML's adoption, data integration systems needed custom wrappers that did "screen scraping" (custom HTML parsing and content extraction) to extract content from XML, and that translated to the proprietary wire formats of different legacy tools. Today, we can expect most sources to have an XML interface (URIs as the request mechanism and XML as the returned data format), and thus the data integrator can focus on the semantic mappings rather than the low-level format issues.

We note that XML brings standardization not only in the actual data format, but also in terms of an entire ecosystem of interfaces, standards, and tools: DTD and XML Schema for specifying schemas, DOM and SAX for language-neutral parser interfaces, WSDL/SOAP or REST for invoking Web services, text editors and browsers with

[1]JSON, the JavaScript Object Notation, can be thought of as a "simplified XML" although it also closely resembles previous *semistructured data* formats like the Object Exchange Model.

built-in support for XML creation, display, and validation. Any tool that produces and consumes XML automatically benefits from having these other components.

This book does not attempt to cover all of the details of XML, but rather to provide the core essentials. In this chapter, we focus first on the XML data model in Section 8.1 and the schema formalisms in Section 8.2. Section 8.3 presents several models for querying XML, culminating in the XQuery standard that is implemented in many XML database systems. We then discuss the subsets of XQuery typically used for XML schema mappings (Section 8.4) and then discuss XML query processing for data integration (Section 8.5).

8.1 Data Model

Like HTML (HyperText Markup Language), XML (eXtensible Markup Language) is essentially a specialized derivative of an old standard called SGML (Structured Generalized Markup Language). As with these other markup standards, XML encodes document meta-information using *tags* (in angle brackets) and *attributes* (attribute-value pairs associated with specific tags).

XML distinguishes itself from its predecessors in that (if correctly structured, or *well-formed*) it is *always parsable* by an XML parser — regardless of whether the XML parser has any information that enables it to interpret the XML tags. To ensure this, XML has strict rules about how the document is structured. We briefly describe the essential components of an XML document.

Processing instructions to aid the parser. The first line of an XML file tells the XML parser information about the character set used for encoding the remainder of the document; this is critical since it determines how many bytes encode each character in the file. Character sets are specified using a *processing-instruction*, such as `<?xml version="1.0" encoding="ISO-8859-1"?>`, which we see at the top of the example XML fragment in Figure 8.1 (an excerpt from the research paper bibliography Web site DBLP, at `dblp.uni-trier.de`). Other processing instructions may specify constraints on the content of the XML document, and we will discuss them later.

Tags, elements, and attributes. The main content of the XML document consists of tags, attributes, and data. XML tags are indicated using angle-brackets, and must come in pairs: for each open-tag `<tag>`, there must be a matching close-tag `</tag>`. An open-tag / close-tag pair and its contents are said to be an *XML element*. An element may have one or more *attributes*, each with a unique name and a value specified within the open-tag: `<tag attrib1="value1" attrib2="value2">`.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<dblp>
  <mastersthesis mdate="2002-01-03" key="ms/Brown92">
   <author>Kurt P. Brown</author>
   <title>PRPL: A Database Workload Specification Language</title>
   <year>1992</year>
   <school>Univ. of Wisconsin-Madison</school>
  </mastersthesis>
  <article mdate="2002-01-03" key="tr/dec/SRC1997-018">
   <editor>Paul R. McJones</editor>
   <title>The 1995 SQL Reunion</title>
   <journal>Digital System Research Center Report</journal>
   <volume>SRC1997-018</volume>
   <year>1997</year>
   <ee>db/labs/dec/SRC1997-018.html</ee>
   <ee>http://www.mcjones.org/System_R/SQL_Reunion_95/</ee>
 </article>
  ...
</dblp>
```

Figure 8.1: Sample XML data from the DBLP Web site

Of course, an element may contain nested elements, nested text, and a variety of other types of content we will describe shortly. It is important to note that every XML document must contain a single *root element*, meaning that the element content can be thought of as a tree and not a forest.

Example 8.1: Figure 8.1 shows a detailed fragment of XML from DBLP, as mentioned on the previous page. The first line is a processing instruction specifying the character set encoding. Next comes the single *root element*, `dblp`. Within the DBLP element we see two sub-elements, one describing a `mastersthesis` and the other an `article`. Additional elements are elided.

Both the MS thesis and article elements contain two attributes, `mdate` and `key`. Additionally, they contain sub-elements such as `author` or `editor`. Note that `ee` appears twice within the `article`. Within each of the sub-elements at this level is *text content*, each contiguous fragment of which is represented in the XML data model by a text node.

Namespaces and qualified names. Sometimes an XML document consists of content merged from multiple sources. In such situations, we may have the same tag names in several sources, and may wish to differentiate among them. In such a situation, we can give each of the source documents a *namespace*: this is a globally unique name, specified in the form of a Uniform Resource Indicator (URI). (The URI is simply a unique name specified in the form of a qualified path, and does not necessarily represent the address of any particular content. The more familiar Uniform Resource Locator, or URL, is a special case of a URI where there is a data item whose content can be retrieved according to the path in the URI.) Within an XML document, we can assign a much shorter name, the *namespace prefix*, to each of the namespace URIs. Then, within the XML document, we can "qualify" individual tag names with this prefix, followed by a colon, e.g., `<ns:tag>`. The *default namespace* is the one for all tags without qualified names.

Document order. XML was designed to serve several different roles simultaneously: extensible document format generalizing and replacing HTML; general-purpose markup language; structured data export format. It distinguishes itself from most database-related standards in that it is *order-preserving* and generally *order-sensitive*. More specifically, the order between XML elements is considered to be meaningful, and is preserved and queriable through XML query languages: this enables, e.g., ordering among paragraphs in a document to be maintained or tested. Perhaps surprisingly, XML *attributes* (which are treated as properties of elements) are *not* order-sensitive, although XML tools will typically preserve the original order.

We typically represent the logical or data model structure of an XML document as a tree, where each XML node is represented by a node in the tree, and parent-child relationships are encoded as edges. There are seven node types, briefly alluded to above:

- **Document root**: this node represents the entire XML document, and generally has as its children at least one processing instruction (representing the XML encoding information) and a single root element.
- **Processing instruction**: these nodes instruct the parser on character encodings, parse structure, etc.
- **Comment**: as with HTML comments, these are human-readable notes.
- **Element**: most data structures are encoded as XML elements, which include open and close tags plus content. A content-free element may be represented as a single *empty tag* of the form `<tag/>`, which is considered equivalent to an open-tag / close-tag sequence.

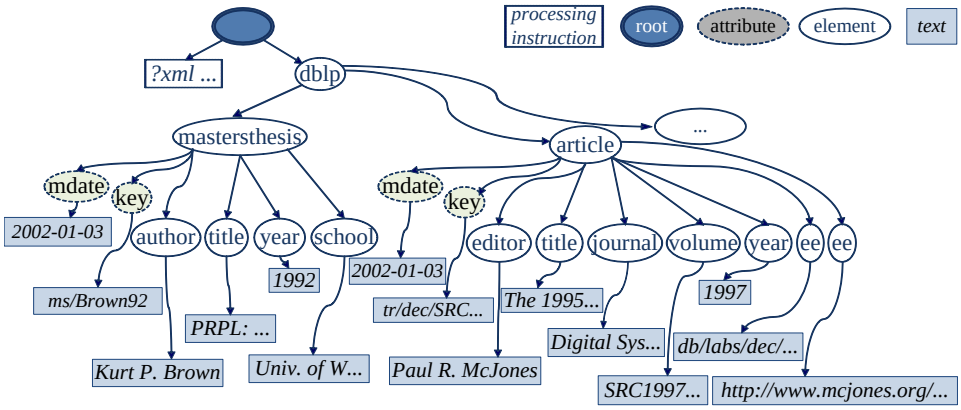

Figure 8.2: Example data model representation for the XML fragment of Figure 8.1. Note that the different node types are represented here using different shapes.

- **Attribute**: an attribute is a name-value pair associated with an element (and embedded in its open tag). Attributes are not order-sensitive, and no single tag may have more than one attribute with the same name.
- **Text**: a text node represents contiguous data content within an element.
- **Namespace**: a namespace node qualifies the name of an element within a particular URI. This creates a *qualified name*.

Each node in the document has a unique identity, as well as a relative ordering and (if a schema is associated with the document) a datatype. A depth-first, left-to-right traversal of the tree representation corresponds to the node ordering within the associated XML document.

Example 8.2: Figure 8.2 shows an XML data model representation for the document of Figure 8.1. Here we see five of the seven node types (comment and namespace are not present in this document).

8.2 XML Structural and Schema Definitions

In general, XML can be thought of as a semi-structured hierarchical data format, whose leaves are primarily string (text) nodes and attribute node values. To be most useful, we must add a *schema* describing the semantics and types of the attributes and elements — this enables us to encode non-string datatypes as well as inter- and intra-document links (e.g., foreign keys, URLs).

```
<!ELEMENT dblp((masterthesis | article)*)>
<!ELEMENT masterthesis(author,title,year,school,committeemember*)>
<!ATTLIST masterthesis(mdate   CDATA  #REQUIRED
                       key     ID     #REQUIRED
                       advisor CDATA  #IMPLIED>
<!ELEMENT author(#PCDATA)>
<!ELEMENT title(#PCDATA)>
<!ELEMENT year(#PCDATA)>
<!ELEMENT school(#PCDATA)>
<!ELEMENT committeemember(#PCDATA)>
 ...
```

Figure 8.3: Fragment of an example DTD for the XML of Figure 8.1. Note that each definition here is expressed using a processing instruction.

8.2.1 Document Type Definitions (DTDs)

When the XML standard was introduced, the focus was on document markup, and hence the original "schema" specification was more focused on the legal structure of the markup than on specific datatypes and data semantics. The *document type definition* or DTD is expressed using processing instructions, and tells the XML parser about the structure of a given element.

In DTD, we specify for each element which sub-elements and/or text nodes are allowed within an element, using EBNF notation to indicate alternation or nesting. A sub-element is represented by its name, and a text node is designated as `#PCDATA` ("parsed character" data).

Example 8.3: Figure 8.3 shows a fragment of a DTD for our running example in this section. We focus in this portion of the example on the element definitions (`ELEMENT` processing instructions). The DBLP element may have a sequence of `masterthesis` and `article` sub-elements, interleaved in any order. In turn the `masterthesis` has mandatory `author`, `title`, `year`, and `school` sub-elements, followed by zero or more `committeemember`s. Each of these sub-elements contains text content.

Attributes are specified in a series of rows within the `ATTLIST` processing instruction: each attribute specification includes a name, a special type, and an annotation indicating whether the attribute is optional (`#IMPLIED`) or mandatory (`#REQUIRED`). The types are as follows: text content is called `CDATA` ("character data"), which is somewhat more restricted than the `PCDATA` allowed in elements; `ID` designates that the

attribute contains an *identifier* that is globally unique within the document; `IDREF` or `IDREFS` specifies that the attribute contains a *reference* or space-separated references, respectively, to `ID`-typed attributes within the document. `ID`s and `IDREF`s can be thought of as special cases of keys and foreign keys, or anchors and links.

Example 8.4: Figure 8.3 defines three attributes related to the `mastersthesis`. The `mdate` is mandatory and of text-type. The `key` is also mandatory, but a unique identifier. Finally, the advisor string is optional.

Oddly enough, the DTD does not specify what element is the root within a document. The root element is instead specified within the processing instruction *in the source XML* that references the DTD. The DTD can be directly embedded within the document using a syntax like:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE dblp [
  <!ELEMENT dblp((mastersthesis | article)*)>
  <!ELEMENT mastersthesis(author,title,year,school,committeemember*)>
  ...
 ]>
<dblp>
 ...
```

but more commonly, the DTD is in a separate file that must be referenced using the `SYSTEM` keyword:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE dblp SYSTEM "dblp.dtd">
<dblp>
 ...
```

In both cases, the first parameter after `DOCTYPE` is the name of the root element, and the XML parser will parse the DTD before continuing beyond that point.

Pros and cons of DTD. DTD, as the first XML schema format, is very commonly used in practice. It is relatively simple to understand and concise (if syntactically awkward), it is supported by virtually every XML parser in existence, and it is sufficient for most document structural specifications. Unfortunately, it has many limitations for data interchange. One cannot directly map the database concepts of key and

foreign key to `ID` and `IDREFS` (there is no support for compound keys, and the value of a key must be *globally unique* within a document, even to the exclusion of `ID`-typed attributes with other names). The concept of null values does not map to any aspect of DTD-based XML, meaning that relational database export is awkward. Primitive datatypes such as integers and dates are not possible to specify.

All of these limitations, as well as a variety of other desiderata, led to the development of a newer specification in the early 2000s, called XML Schema.

8.2.2 XML Schema (XSD)

XML Schema (commonly abbreviated to its standard 3-letter file extension, XSD) is an extremely comprehensive standard designed to provide a superset of DTD's capabilities, to address the limitations mentioned in the previous section, and to itself be an XML-encoded standard.

Since an XML Schema is itself specified in XML, we will always be dealing with (at least) two namespaces: the namespace for XML schema itself (used for the built-in XSD definitions and datatypes), and the namespace being defined by the schema. Typically, we will use the default namespace for the tags defined in the schema, and will use a prefix (commonly `xs:` or `xsd:`) associated with the URI `www.w3.org/2001/XMLSchema` to refer to the XML Schema tags.

Beyond the use of XML tags, XML Schema differs significantly from DTD in two respects. First, the notion of an *element type* has been separated from the *element name*. Now we define an element type (either a `complexType` representing a structured element, or a `simpleType` representing a scalar or text node) and later associate it with one or more element names. Second, the use of EBNF notation has been completely eliminated, and instead we group sequences of content using `sequence` or `choice` elements, and specify a number of repetitions using `minOccurs` and `maxOccurs` attributes.

Example 8.5: Figure 8.4 shows an XML Schema fragment for our running example. We see first that the schema definition has a root tag `schema` within the XML Schema namespace (abbreviated here as `xsd`).

The fragment shows the definition of a complex element type for a thesis. Associated with this element type are three attributes (`mdate`, `key`, and optional `advisor`). Observe that each of the attributes has a particular type (one of the built-in XML Schema *simple types*): date, string, and string, respectively. Within the `ThesisType` is a sequence of subelements: an `author` string, a `title` string, a `year` integer, a school `string`, and a sequence of zero or more `committeemember`s of a complex type

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 ...
 <xsd:complexType name="ThesisType">
  <xsd:attribute name="mdate" type="xsd:date"/>
  <xsd:attribute name="key" type="xsd:string"/>
  <xsd:attribute name="advisor" type="xsd:string" minOccurs="0"/>
  <xsd:sequence>
   <xsd:element name="author" type="xsd:string"/>
   <xsd:element name="title" type="xsd:string"/>
   <xsd:element name="year" type="xsd:integer"/>
   <xsd:element name="school" type="xsd:string"/>
   <xsd:element name="committeemember" type="CommitteeType"
                minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
 </xsd:complexType>
 <xsd:complexType name="CommitteeType">
  ...
 </xsd:complexType>
 ...
 <xsd:element name="mastersthesis" type="ThesisType"/>
 ...
</xsd:schema>
```

Figure 8.4: Example XML Schema fragment corresponding to the XML of Figure 8.1. This excerpt focuses on defining the structure of the `mastersthesis`.

called `CommitteeType`. The `ThesisType` can be used for more than one element; we finally associated it with the `mastersthesis` near the bottom of the figure.

XML Schema allows for the definition of both keys and foreign keys, which we shall discuss later in this chapter when we have introduced XPath. It also has many features that we do not discuss at all: it is possible to define simple types that restrict the built-in types (e.g., positive integers, dates from 2010-2020, strings starting with "S"), to make use of inheritance in defining types, and to create reusable structures. For more detail we suggest consulting the many resources available on the Web.

XML Schema and its associated data model are the "modern" schema format for XML, and used by most XML-related Web standards. Examples include SOAP, the Simple Object Access Protocol used to pass parameters across systems; WSDL, the Web Service Description Language; the datatype primitives of RDF Schema, the

schema language for the Resource Description Framework used in the Semantic Web (see Section 12.3.3); and XQuery, the XML query language we describe later in this chapter.

8.3 Query Language

Given that XML represents documents and data in a standard way, it quickly became apparent that applications and developers would need standardized mechanisms for extracting and manipulating XML content.

Over the years, models emerged for parsing XML into objects (DOM) and messages (SAX), the XPath primitive, and ultimately the XQuery language. We discuss the parsing standards in Section 8.3.1, XPath in Section 8.3.2, and XQuery in Section 8.3.3. Another XML transformation language, XSLT (XML Stylesheet Language: Transformations), is used in many document and formatting scenarios, but is not well-suited to data manipulation and hence we do not discuss it in this book.

8.3.1 Precursors: DOM and SAX

Prior to the creation of XML query primitives, the XML community established a series of language-independent APIs for parsing XML: the goal was that any parser would implement these APIs in a consistent way, making it easy to port code to a variety of platforms and libraries.

The first standard, the Document Object Model or DOM, specifies a common object-oriented hierarchy for parsed HTML and XML. DOM actually emerged from the internal object models supported by HTML web browsers, but it was generalized to XML as well. DOM establishes a common interface, the DOM Node, to encompass all of the various XML node types; instances include, e.g., the Document Node, Element Node, and Text Node. A DOM parser typically builds an in-memory object tree representing a parsed document, and returns the document root node to the calling program. From here the application can traverse the entire document model: Every DOM node has methods for traversing to the node's parent and children (if applicable), testing the node type, reading the text value of the node, and so on. An interface also exists to retrieve nodes directly by their name, rather than through the document hierarchy. Later versions of DOM also support *updates* to the in-memory document model.

DOM is a fairly heavyweight representation of XML: each node in an XML document is instantiated as an object, which typically consumes significantly more space than the original source file. Moreover, early versions of DOM were not in any way

incremental: no processing would happen until the entire document was parsed. To meet the needs of applications that only wanted to manipulate a small portion of an XML document — or to incrementally process the data — the SAX, the Simple API for XML, was created. SAX is not an object representation, but rather a standard parser API. As an XML parser reads through an input XML file, it *calls back* to user-supplied methods, notifying them when a new element is beginning, when an element is closing, when a text node is encountered, etc. The application can take the information provided in the callback, and perform whatever behaviors it deems appropriate. The application might instantiate an object for each callback, or it might simply update progress information or discard the callback information entirely.

Both SAX and DOM are designed for developers in object-oriented languages to manipulate XML; they are not declarative means for extracting content. Of course, such declarative standards have also been developed, and we discuss them next.

8.3.2 XPath: A Primitive for XML Querying

XPath was originally developed to be a simple XML query language, whose role is to extract subtrees from an individual XML document. Over time it has become more commonly used as a building block for other XML standards: e.g., it is used to specify keys and foreign keys in XML Schema, and it is used to specify collections of XML nodes to be assigned to variables in XQuery (described below).

XPath actually has two versions, the original XPath 1.0 and the later XPath 2.0. The original version was limited to expressions that did not directly specify a source XML document (i.e., one needed to use a tool to apply the XPath to a specific XML file). Version 2.0 was revised during the development of XQuery, in order to make it a fully integrated subset of that language: it adds a number of features, with the most notable being the ability to specify the source document for the expression, and a data model and type system matching that of XQuery. As of the writing of this book, many tools still use the original XPath 1.0 specification.

Path expressions. The main construct in XPath is the *path expression*, which represents a sequence of *steps* from a *context node*. The *context node* is by default the root of the source document to which the XPath is being applied. The result of evaluating the path expression is a *sequence of nodes* (and their subtree descendants), with duplicates removed, returned in the order they appear in the source document. (This sequence is often termed a "node set" for historic reasons.)

The context node. XPaths are specified using a Unix path-like syntax. As in Unix, the *current* context node is designated by "." If we start the XPath with a

leading "/" then this starts at the document root. In XPath 2.0, we can also specify a particular source document to use as the context node: the function `doc("URL")` parses the XML document at *URL* and returns its document root as the context node.

From the context node, an XPath typically includes a sequence of *steps* and optional *predicates*. If we follow the usual interpretation of an XML document as a tree, than a step in an XPath encodes a *step type* describing how to traverse from the context node, and a *node restriction* specifying which nodes to return. The *step-type* is typically a delimiter like "/" or "//" and the *node-restriction* is typically a label of an element to match; we specify these more precisely in a moment.

By default, traversal is downwards in the tree, i.e., to descendants. We can start at the context node and traverse a *single level* to a child node (specified using the delimiter "/") or through *zero or more descendant sub-elements* (specified using "//"). We can restrict the matching node in a variety of ways:

- A step "$\cdots$`/label`" or "$\cdots$`//label`" will only return child or descendant elements, respectively, with the designated label. (Any intervening elements are unrestricted.)
- A step with a "`*`" will match any label: "$\cdots$`/*`" or "$\cdots$`//*`," will return the set of all child elements, or descendant elements, respectively.
- A step "$\cdots$`/@label`" will return attribute nodes (including both label and value) that are children of the current element, which match the specified label; the step or "$\cdots$`//@label`" will return attribute nodes of the current *or any descendant element*, if they have the specified label.
- A step "$\cdots$`/@*`" or "$\cdots$`//@*`" will return any attribute nodes associated with the current element, or the current and any descendant elements, respectively.
- A step "`/..`" represents a step up the tree to the *parent* of each node matched by the previous step in the XPath.
- A step with a *node-test* will restrict the type of node to a particular class: e.g., $\cdots$`/text()` returns child nodes that are text nodes; $\cdots$`/comment()` returns child nodes that are comments; $\cdots$`/processing-instruction()` returns child nodes that are processing instructions; $\cdots$`/node()` returns any type of node (not just elements, attributes, etc.).

Example 8.6: Given the data of Figure 8.1, the XPath `./dblp/article` would begin with the document root as the context node, traverse downward to the `dblp` root element and any `article` subelements, and return an XML node sequence containing

the `article` elements (which would still maintain all attachments to its subtree). Thus, if we were to serialize the node sequence back to XML, the result would be:

```
<article mdate="2002-01-03" key="tr/dec/SRC1997-018">
 <editor>Paul R. McJones</editor>
 <title>The 1995 SQL Reunion</title>
 <journal>Digital System Research Center Report</journal>
 <volume>SRC1997-018</volume>
 <year>1997</year>
 <ee>db/labs/dec/SRC1997-018.html</ee>
 <ee>http://www.mcjones.org/System_R/SQL_Reunion_95/</ee>
</article>
...
```

(Note that if there were more than one `article`, the result of the XPath would be a *forest* of XML trees, rather than a single tree. Hence the output of an XPath is often not a legal XML document since it does not have a single root element.)

Example 8.7: Given the example data, the XPath `//year` would begin at the document root (due to the leading "/"), traverse any number of subelements downward, and return:

```
<year>1992</year>
<year>1997</year>
...
```

where the first `year` comes from the `mastersthesis` and the second from the `article`.

Example 8.8: Given the example data, the XPath `/dblp/*/editor` would begin at the document root, traverse downwards to match first the `mastersthesis` and look for an `editor`. No such match is found, so the next traversal will occur in the `article`. From the `article`, an `editor` can be found, and the result would be:

```
<editor>Paul R. McJones</editor>
...
```

where, of course, further results might be returned if there were matches among the elided content in the document.

More complex steps: axes. While they are by far the most heavily used step specifiers, "`/`" and "`//`" are actually considered to be a special *abbreviated syntax* for a more general XML traversal model called *axes*. Axes allow an XPath author to not only specify a traversal to a descendant node, but also an ancestor, a predecessor, a successor, etc. The syntax for axes is somewhat cumbersome: instead of using "`/`" or "`//`" followed by a node restriction, we instead use "`/`" followed by an *axis specifier* followed by "`::`", then the node restriction. The axes include:

- `child`: traverses to a child node, identically to a plain "`/`".
- `descendant`: finds a descendant of the current node, identically to a plain "`//`".
- `descendant-or-self`: returns the current node or any descendant.
- `parent`: returns the parent of the current node, identically to "`..`".
- `ancestor`: finds any ancestor of the current node.
- `ancestor-or-self`: returns the current node or an ancestor.
- `preceding-sibling`: returns any sibling node that appeared earlier in the document order.
- `following-sibling`: returns any sibling node that appeared later in the document order.
- `preceding`: returns any node, at any level, appearing earlier in the document.
- `following`: returns any node, at any level, appearing later in the document.
- `attribute`: matches attributes, as with the "" prefix.
- `namespace`: matches namespace nodes.

Example 8.9: The XPath of Example 8.6 could be written as:

```
./child::dblp/child::article
```

and Example 8.7 as:

```
/descendant::year
```

An XPath to return *every* XML node in the document is:

```
/descendant-or-self::node()
```

Predicates. Often we want to further restrict the set of nodes to be returned, e.g., by specifying certain data values we seek. This is where XPath *predicates* come in. A predicate is a Boolean test that can be attached to a specific step in an XPath; the Boolean test is applied to each result that would ordinarily be returned by the XPath. The Boolean test may be as simple as the existence of a path (this enables us to express queries that test for specific *tree patterns* as opposed simply to paths), it could be a test for a particular value of a text or attribute node, etc.

Predicates appear in square brackets, `[` and `]`. The expression within the brackets has its context node set to the node matched by the previous step in the XPath.

Example 8.10: If we take the XPath expression `/dblp/*[./year/text()="1992"]` and evaluate it against Figure 8.1, then first `mastersthesis` is matched (becoming the context node becomes the `mastersthesis`), and then the predicate expression is evaluated. Since it returns true, the `mastersthesis` node is returned. When the `article` node is matched (becoming the context node for the predicate), the predicate is again evaluated, but this time it fails to satisfy the conditions.

Predicates referring to position. Predicates can also be used to select only specific values from a node set, based on the index position of the nodes. If we use an integer value i as the position, e.g., `p[5]`, this selects the ith node in the node sequence. We can also explicitly request the index of a node with the `position()` function, and the index of the *last* node in the node sequence with the `last()` function.

Example 8.11: To select the last article in the XML file of Figure 8.1, we could use the XPath:

```
//article[position()=last()]
```