Chapter 2

Overview of XML

In this chapter we give a brief introduction to XML, its context and the role it plays in a Web environment. We briefly discuss the syntax of XML documents and the semantics it can represent. Since we do not have the space to exhaust all details of the technical definition of XML, we restrict ourselves to the presentation of the most prominent features that form the backbone of XML and make it a challenge for database management systems. Readers who are familiar with XML and the standard infrastructure may want to skip this chapter.

2.1 What is XML?

When asked for a concise definition of XML, one could say that XML is a markup language for structured documents. It has been standardised by the World Wide Web Consortium (W3C [W3C01a]) and serves as a syntactical framework for data interchange on the Internet. Because it was backed by the leading players in the software and communications industry, it quickly assumed the role of the ubiquitous and universal data interchange format.

Structured documents are documents which not only contain content particles, i.e., plain text, but also feature syntactic annotations of what the semantic relationships between these particles are. XML gives the user two basic tools to structure documents: elements and attributes. Elements are named lists which again contain elements or text; each element carries a name, called tag, and an association list of attributes; i.e., a list that consists of name-value (attribute name – attribute value) pairs. The XML standard defines two special kinds of attributes: identifiers (IDs) and identifier references (IDREFs); XML parsers also have to check certain semantic constraints that the presence of these attributes may require. Identifiers have to be unique throughout the documents, i.e., no two elements may carry the same ID, and identifier references must point to identifiers which exist in the document; otherwise, a parser is supposed to issue an error message in this case and abort. It should be noted that for most documents there is more than one natural way to group content particles, an issue which will be discussed later in more detail.

A markup language is a mechanism to single out the document structure from the content particles usually by employing two different but not necessarily disjoint alphabets. One of the main contributions of XML is to provide a standard way of doing this, thus enabling the user of one parser for many different applications. Historically,
XML can be seen as a mixture of regular right-part grammars and parenthesis grammars [GH67, LaL77].

It is linguistically unfortunate that marked-up documents are called structured documents whereas the graph representation of those documents is called semi-structured data. In this sense we can say that, for our purposes, semi-structured data are abstract representations of structured documents. Some database researchers also call structured documents semi-structured documents to highlight the difference between the flexible structure of documents and the very regular nature of (relational) databases.

To illustrate the previous concepts, we now start with the presentation of a running example. Historically, bibliographies are one of the favourite instances of semi-structured documents since they come from a domain, namely libraries, with which readers usually are familiar. As an example, consider the snippet from a bibliography (we follow the conventions of the ACM SIGMOD bibliography [Ley99], which again follows the BuTeX [Lam86] file structure; other bibliographic formats like RFC 1807 [LC95], Refer [Cla89], Tib [Ale86], MARC [LibOl] and others are of similar structure; so the example indeed could be found in a real site) displayed in Figure 2.1. The indentation of the document suggests that XML data form a tree, and, indeed, every well-formed XML documents possesses exactly one root. We call an XML document

Figure 2.1: A bibliography as an instance for a semi-structured document
2.1 What is XML?

```
<article key="BB88" rating="excellent">
  <author>
    <first>Ben</first>
    <last>Bit</last>
    <title>How To Hack</title>
    <year>1988</year>
  </author>
</article>
```

First alternative formulation

```
<article>
  <key>BB88</key>
  <rating>excellent</rating>
  <author>Ben Bit</author>
  <title>How To Hack</title>
  <year>1988</year>
</article>
```

Second alternative formulation

Figure 2.2: Reformulations of the first article of Figure 2.1

well-formed if it adheres to a tree structure and obeys other syntactic and semantic restrictions which we will mention when it is appropriate.

So, XML documents comprise text and tags. In the example, Ben Bit, How to Hack, or 1988 constitute the text part, whereas <bibliography, <article ...> are tags: tags are enclosed in angle brackets (<>), which must not appear as such in text, and may carry an association list. The first element of tags (article, author, ...) is also called element or element name. Each element may carry an association list of name-value or symbol-value pairs, called attributes; for example, the association list of the first article element assigns the value BB88 to the symbol key and the value excellent to the symbol rating. Values are required to consist of the same alphabet as element names. Hence, sub-elements and attributes are not freely interchangeable, since the latter can't have a complex internal structure.

There are a number of constraints not only on the structure of documents but also on the contents: in the case of attributes, names have to be unique per attribute list just like association lists in programming languages. Furthermore, if the elements are properly nested, i.e., there is no pair of opening and closing tags that spans more than one subtree like <a><b></a></b>, then the two most fundamental constraints for well-formed documents are met. XML parsers are expected to reject documents which are not well-formed. Elements can also be empty; then <a></a> can be abbreviated as <a/>. Note that the offspring of a tag is actually an ordered sequence of elements and text and not a set of associations, which would be unordered.

It should be noted that the presence of attributes and elements in the language definition concedes modelling freedom to the document designer, who can often choose between using one of two. In Figure 2.2 we present reformulations of the first article record of the example document in Figure 2.1. In the first alternative, semantics have been added since the author name is split up into first and last name. In the second alternative, the attributes 'key' and 'rating' have been converted to elements; there
is an argument whether such transformations change the semantics of the document [Moe00]. Other transformations are possible; however, in this thesis we assume that different markup structures imply different documents, i.e., that the document designer has consciously chosen a certain strategy to markup data.

Coming back to XML documents and our example, we would like to mention entities, which are either abbreviations (see later in this section where we describe DTDs) or representations for characters which are not legal in XML documents. XML entities, unlike SGML entities, do not extend the expressiveness of the language; Figure 2.3 lists some of the most common ones. The title of the second article record in the example document in Figure 2.1 also contains the entity &amp; to represent the character &, which is not legal in XML documents.

Having introduced these concepts, we can now define the notion of an XML document:

**Definition 1.** Let $T$ be an alphabet of opening brackets (‘tags’) and $\bar{T}$ an alphabet of closing brackets (note that there has to exist a bijection between $T$ and $\bar{T}$). Let $\Sigma$ be an alphabet (for ‘text’) and $S = \Sigma \cup (T \cup \bar{T})$. Then an XML document $X$ is defined recursively as follows: $X \rightarrow (t, (S \cup X)^*, \bar{t}, \alpha)$, where $^*$ is the Kleene star and $t \in T, \bar{t} \in \bar{T}$ and $\alpha : \Sigma^* \rightarrow \Sigma^*$, a function which assigns attributes, i.e., a key-value pairs, to every derivation every derivation.

Although we do not make use of this definition intensively, it defines the way XML documents are viewed in this thesis: we abstract from the textual representation or parsing issues like entity expansion and focus on the tree shape of documents. This has implications, for example, on the way we store documents or how we define the notion of proximity between syntactic elements.

**Example 2.** The example document now can be described as follows:

1. $T = \{\text{bibliography, article,}\ldots\}$,
2. $\bar{T} = \{/\text{bibliography, /article,}\ldots\}$,
3. $\Sigma = \{A,\ldots,Z,a,\ldots,z,\ldots\}$,
4. (bibliography,
   
   (article,
   
   (author,
   
   Ben Bit,
   
   /author,\})
   
   (title,
This representation is very similar to how XML documents would be represented with nested list structures in functional programming languages like Haskell [WR99] or Lisp [Wad99].

We now introduce additional features of XML which were not included into the abstract model because they may be expressed by elements and attributes in combination with application specific knowledge, which they would require anyway when they are evaluated. Notations are a way to link entities in the documents to document-external entities by means of Uniform Resource Identifiers (URI) [BLFIM98]. Processing Instructions are used to pass information to external applications, style-sheets are a popular example, and determine their behaviour. Comments finally are used to annotate documents for human readers; they should not influence the behaviour of (automated) applications.

XML not only leaves structural freedom to the document designer, but also often syntactical freedom is an issue, too. In our example, the first article record encloses text in elements without adding spaces whereas the second separates authors' names and tags with a single whitespace; the third record even introduces line breaks and also employs additional tags. Eventually, applications which consume the documents have to deal with the representational heterogeneity but one problem remains even on the specification level. It is not clear when to regard two XML documents as equivalent in a general way. Clearly, a byte-to-byte comparison does not make sense since even the standard specifies equivalences, e.g., as we have seen, <a/> is equivalent to <a></a> (and, by the way most XML parsers like [ea01] resolve the first case to the second). To this end, canonical XML [Boy01], a subset of XML, was introduced to resolve ambiguities and place constraints like lexicographic ordering of attributes on the canonical version of the document and, this way, to define some kind of normal form. Every well-formed XML document has a unique structurally equivalent canonical XML document. If two XML documents are structurally equivalent, then their canonical versions are byte-for-byte identical. This is one way to compare documents without application specific knowledge. Canonicalising an XML document requires only information that an XML parser or processor is required to make available to an application anyway, so for example the order of attributes does not contribute to the notion of isomorphic documents. Especially, ignorable white space is treated like text data. We therefore say that two XML documents are isomorphic if their canonical representations are equal. Of course, applications may provide their own notion of equality if this is appropriate.

### 2.2 Constraints on Documents

In this section, we motivate the need to define constraints on documents and give an overview of the principles of existing languages being used to define them. In application scenarios, these languages present document-external application-specific
knowledge from which we can draw benefit when we use databases to process XML. Constraint definition languages also bear similarities with Data Definition Languages (DDL) which are used in relational database management systems [Ame86] and object-oriented programming languages [Str86].

2.2.1 Defining and Using Constraints

Since one of XML's most important purposes is to serve as a data interchange language, one approach to making communication more reliable is to formalise and publish knowledge that applications have about the network traffic they produce and consume. This knowledge can then be used to validate messages before they are consumed or to verify code that emits messages. Furthermore, schema information can reduce the amount of fuzziness that is introduced by the textual layers in data interchange, for example by classifying which documents belong to the class of documents that is permissible as input to an algorithm; ideally, this classification is done automatically by the machines involved and without human intervention. Most schema languages take the element relationships, i.e., the tree-structured part of XML, as the basis of the constraint definition language. This way it is possible to describe both complete document trees but also sub-trees, i.e., partial documents. If a document instance meets all constraints specified in a schema document we say that it conforms to a schema or that it is valid.

The most basic kind of validation is included in the XML standard: the specification language called Document Type Definition (DTD) is inherited by XML from its SGML ancestry and is a proper subset of the modelling construct that SGML DTDs provide. They will be presented in more detail in the following subsection. Unfortunately plain DTDs do not offer the expressiveness that many application programmers would like to have at their hands. Standardisation committees and researchers therefore set out to transfer ideas from related areas like logic programming or formal language theory to document processing and experimented with new ideas in schema languages. One of the requirements was to make the schema definitions themselves accessible to XML query languages and use XML syntax to express the constraints, which was not true for DTDs since they have their own syntax which is not based on XML. Other key ideas were to provide ways to extend schemas and a cleaner separation between document parsing and schema validation, which are very much intertwined in the case of DTDs. An early example of a schema definition language is DDML, previously known as XSCHEMA [W3C99], which served as a starting point for many other languages. It was later superseded by XML Schema, the W3C's family of schema languages based on XML syntax which currently consists of three individual languages. Each of those languages focuses on a particular aspect of syntactic and semantic constraints such as document structure [W3C01e] or pre- and user-defined data types [W3C01f]. To make DDML specifications accessible to legacy software, tools were provided to convert the then already standardised DTDs to DDML syntax and vice versa.

Technically, XML schema languages are designed to align with design and maintenance cycles of software systems. In this sense they are very much similar to DDLs in Relational Databases and, consequently, have an overall influence on API design: their big plus is that they make knowledge explicit, which normally would have been hidden and encoded in program-intenal data structures and the functions that operate on them. This transparency eventually helps software architects keep up with a changing world; they make it easier to trace how changes propagate through all layers of a complex application.
And indeed, some of the practical implications and benefits of schema languages have become apparent. Schema specification is always part of a software engineering process; the vocabulary provided can serve as a *road map* for consistent implementation of XML-based communication procedures. This also helps to cope with the many degrees of freedom that XML concedes to the users, which on the one hand is a primary reason for its popularity; on the other hand, without means to eliminate semantic ambiguities this freedom would tend to degenerate into vagueness that can make it difficult to pin down the exact behaviour of an application for example in an interface specification. By combining a precise schema definition with a specification, XML can enable advancements in application integration. For example, the BizTalk Framework [MLM+01] “establishes a set of rules that enables a broad audience to adopt a common approach to using XML.” It does so by installing a core set of XML element and attribute structures that enable companies to implement their business processes in this framework and draw benefit from an optimised and simplified messaging framework. The definition of a core set is considered important because the XML patterns form the glue that binds systems only connected by a wire(less) network, and thus eliminates the need to find a common application programming interface (API) or even implementation platform. The primary task of extracting information from the messages is left to platform- and infrastructure-independent high-level XML query languages.

Once schemas have been designed for a number of domains, the need to facilitate and even automate full or partial *mappings between schemas* quickly becomes apparent. The meta-knowledge provided by a schema specification can help to convert data from one format into another, hence from one business process to another. Furthermore the ability to map data easily fosters open standards and interoperability by accelerating development.

While the discussion up to now has focused on the software development process of a single product, schemas furthermore offer a benefit that goes beyond a single development group: they also are straightforward *design targets* for software vendors in general. The high degree of standardisation in the XML world allows service providers to publish programming language independent interfaces in a consistent format. The BizTalk Framework, for example, provides a clear design target for tools and infrastructure for independent software vendors (ISVs) who set out to build the next generations of electronic commerce and application-integration products. Furthermore, schemas lower the entrance fee for *standardisation efforts* by enabling interested parties to concentrate on high-level features. The low-level machinery is provided by the already standardised XML tools. The ubiquity of XML-based electronic data exchange frameworks makes it easy to migrate existing sets of industry interchange standards to new ones and thereby strengthen the robustness and reliability of the software.

Eventually, *consistency* between implementations and versions is ensured through centralised services which are required to provide authoritative versions of the schemas and global objects. Coming back to BizTalk, the framework allows either public or community-centred publication at the authors’ choice. This makes it easy to establish security and encapsulation levels so that objects and users can only access information relevant to them.

### 2.2.2 Document Type Definitions

Document Type Definitions (DTDs) are the most basic form of constraints. They look like Regular Right Part Grammars [LaL77], while they are actually a combination of bracketed grammars [GH67] and Regular Right Part Grammars where the brackets are
<!ELEMENT bibliography (article, inproceedings)*>
<!ELEMENT article (editor?, author*, title, year)>
<!ATTLIST article key CDATA #REQUIRED
rating CDATA #IMPLIED>
<!ELEMENT inproceedings (author*, title, year)>
<!ELEMENT editor (#PCDATA)>
<!ELEMENT author (#PCDATA | (first, last))>
<!ELEMENT title (#PCDATA)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT first (#PCDATA)>
<!ELEMENT last (#PCDATA)>

Figure 2.4: A DTD for the example document of Figure 2.1

implied.

[47] children ::= (choice | seq) ('?' | '*' | '+')?
[48] cp ::= (Name | choice | seq) ('?' | '*' | '+')?
[50] seq ::= '(' S? cp ( S? ',' S? cp )* S? ')'
<!-- Name = tag or character data; S = whitespace -->

Figure 2.5: A snippet from the grammatical specification of DTDs

DTDs are quite limited in their expressiveness. For elements, they allow the specification of which attributes they may carry and which elements are permissible as children. This is done via a system of Backus-Naur like rules. Figure 2.4 shows a DTD that could be used to validate the example bibliography used throughout this chapter; there are two basic types of rule: if they look like <!ELEMENT x Y>, they declare which sequences of tags Y are permissible as children of an element x. Figure 2.5 shows a snippet from the XML standard and explains the structure of Y. It shows that the basic structure of the right side of a production is a recursive nesting of alternatives (choices) and sequences which ends in tags (names) as terminal symbols. Each level of nesting may be attributed with a question mark denoting that there may either be one or zero occurrences of the pattern; accordingly, a plus sign implies one or more occurrences and a (Kleene) star zero or more.

For attributes, DTDs only allow the specification of whether the attribute value may be any given string, be chosen from a set of system-defined values or be supplied by the reading application. Additionally, attributes can be declared required, optional or of fixed content. DTDs also provide a simple notion of data type: the most important are entities, identifiers and identifier references. An XML parser has to reject a document if it encounters an undeclared entity, a non-unique identifiers or a reference which does not point to an identifier in the same document.

2.2.3 XML Schema

Although DTDs have served well for twenty years as the primary mechanism of describing structured information in the SGML and HTML communities, they are con-
2.2 Constraints on Documents

<element name='bibliography' type='Bibliography'/>
<complexType name='Bibliography'>
  <element name='article'
    type='ArticleType'
    minOccurs='0' maxOccurs='unbounded'/>
  <element name='inproceedings'
    type='InproceedingsType'
    minOccurs='0' maxOccurs='unbounded'/>
</complexType>
</element>
<element name='article' type='ArticleType'/>
<complexType name='ArticleType'>
  <attribute name='key' type='string' use='required'/>
  <attribute name='rating' type='string'
    use='optional'/>
  <element name='editor' type='string' minOccurs='0'/>
  <element name='title' type='string'/>
  <element name='author' type='string' minOccurs='1'/>
  <element name='year' type='gYear'/>
</complexType>
</element>
<element name='inproceedings' type='Inproceedings'/>
<complexType name='InproceedingsType'>
  <element name='title' type='string'/>
  <element name='author' type='string' minOccurs='1'/>
  <element name='year' type='gYear'/>
</complexType>
</element>

Figure 2.6: An XML Schema for the example document

Figure 2.6 shows an XML Schema for the example document. Clearly, XML Schema is a more complex description language than DTDs and offers more control over the document structure and text data by introducing types such as generic string but also more specific ones like gYear for Gregorian year. Since XML Schema is too complex to give even an overview of all its features here, we just mention some general principles of the language. A readable schema primer can be found at [W3C01d].

Predefined Types. The standard provides a set of commonly used so-called simple types and allows the definition of complex types which are composed of simple types. Regular expressions, list and set constructs even permit sophisticated structures in text types.

Type Inheritance. XML Schema encourages the reuse of previously defined struc-
tures, no matter whether they are user-defined or pre-defined in the standard. Subtypes can add more elements to a supertype but also represent a subset of values. Groups of attributes and elements can be named for later re-use.

**Controlled Extensibility.** There are mechanisms to define whether a subtype can be derived from a given type but also whether some type can be substituted by another one. There is even the notion of abstract types that must be substituted and of equivalence types.

**Documentation.** To allow for specification of both domain-specific and application-specific knowledge, XML Schema provides dedicated elements like appInfo, documentation and annotation elements for annotating schemas for both human readers and machines.

**Uniqueness Constraints, Keys and References.** It is also possible to declare uniqueness constraints on certain attributes of child elements. This mechanism enables keys and references, too.

**Namespaces.** Sometimes it is desirable that documents conform to more than one schema. To achieve this, XML Schema contains the necessary tools to enable fine-grained control over namespaces.

The process of establishing whether a document conforms to a schema is called *schema validation*. Note that even despite the terminology used, XML Schema is not so much about defining data types like integers or zip codes and hence defining semantics but more about restricting documents so that they the set of permissible (parsed) character data in a content particle looks exactly like an integer or zip code. It does not assign semantics to documents. Data types in the XML world are only introduced by query languages or when documents are processed with programming languages.

### 2.2.4 Other Schema Languages

There are many other efforts to define schema languages. Some are not connected to XML Schema and DTDs, others try to complement or extend them. Within the aforementioned BizTalk framework, the language XDR (XML-Data Reduced schemas) [LJEM+98] is used to describe data. SOX, short for Schema for Object-Oriented XML [DFH+99], aims at improving the integration and accessibility of XML documents in object-oriented programming languages. A unique effort is Schematron [Jel01]; its focal point is validation of schema information rather than the definition of constraints. DSD [KMS00] is an effort to allow context-dependent descriptions of XML documents via constraints. They all differ in ease-of-use and expressiveness. Whereas DTDs are the least expressive and useful constraint language, XDR and SOX substantially improve on them. The three most expressive languages, however, are XML Schema, DSD and Schematron.

### 2.2.5 Inferring Constraints

An interesting question is whether it is possible to infer the grammatical structure of documents from one or more instances. As [Gol67] points out, this in general is not possible. However, practitioners [FX93] sometimes are interested in exploiting document-specific knowledge for storage and query optimisation without having access to the actual or implicit schema information. There are a number of systems that
2.3 Processing XML Documents

try to recognise common patterns implementing different methods; [Lee96] gives an overview of data-mining techniques for this purpose.

2.3 Processing XML Documents

There are two major standardised ways for users to get access to the content of XML documents: event-based parsing APIs and tree-based APIs. While the former enable extremely resource-efficient processing of documents, the latter provide a natural view of documents which is convenient when higher-level applications like query languages are implemented. We will need the two later when we discuss efficient ways to bulkload XML documents and declarative ways to query them.

2.3.1 Event-Based Parsing

An event-based API reports parsing events, such as the start and end of elements, directly to a host application through callbacks of user-registered functions for the different events, and does not usually build an internal parse tree. The Simple API for XML (SAX) [Meg01] is the best known example of event-based parsing and is a de facto standard.

The first step of a SAX parser usually consists of splitting up the source document into tokens. The simple syntax of XML does not require a sophisticated lexer to do this. The most basic way to tokenise a document is to use the occurrences of the brackets < and > as an orientation. For our example document in Figure 2.1, a tokenisation could produce the following stream:

1. "<bibliography>
2. "
3. "<article key="BB88" rating="excellent">
4. "
5. "<author>
6. "Ben Bit"
7. "</author>
8. "
9. "<title>How To Hack"
10. "</title>
11. ...

In general, these token streams events are not immediately suited for digestion in databases. For example, a SAX parser might choose to return "Ben Bit" as two tokens 'Ben B' and 'it'. The author conjectures that this is to make parsing faster and easier. Additionally, SAX parsers de-entitise documents and also tokenise processing instructions etc. Furthermore, the programmer has some control over low-level features like character sets used in the document. Before parsing can begin, users have to register callback functions with the parser. On encountering a token of a specific type, say a start token, the parser calls the function the user previously registered for, in this case, processing start tokens.
package org.w3c.dom;

public interface Document extends Node {
    public DocumentType getDoctype();
    public DOMImplementation getImplementation();
    public Element getDocumentElement();
    public Element createElement(String tagName)
        throws DOMException;
    public DocumentFragment createDocumentFragment();
    public Text createTextNode(String data);
    public Comment createComment(String data);
    public CDATASection createCDATASection(String data)
        throws DOMException;
    public ProcessingInstruction createProcessingInstruction(String target,
        String data)
        throws DOMException;
    public Attr createAttribute(String name)
        throws DOMException;
    public EntityReference createEntityReference(String name)
        throws DOMException;
    public NodeList getElementsByTagName(String tagname);
}

Figure 2.7: Snippet from the DOM specification

2.3.2 Tree Representations

Tree-based APIs convert the document to an internal tree structure. Applications can then navigate through this tree structure via a standard interface. For most navigating applications, the Document Object Model (DOM) [W3C98a] is the API of choice. Often, tree-based APIs use schema information to spare the user from having to write code that dispatches the program control flow according to the element type of the current node and, therefore, allow more declarative programming styles.

While the event-based and stream-oriented perspective of XML documents represents the lowest logical layer in XML processing, tree representations provide more intuitive views. Tree-based XML processors are usually built on top of stream-based processors. The ultimate goal of the DOM standard is to provide a programmatic and language-neutral interface for XML and HTML, which means that the same interface can be used to access documents from different programming languages. The DOM specification comes in three parts: Core, HTML, and XML. The first layer, the Core DOM, provides base classes that can be used to represent parse trees of any structured document written in any markup language. Like stream-based representations, this layer can represent any document in a generic way; the API it defines is minimal and compact and used to navigate through and manipulate the document content generically. However, to support more specific and adaptive programming, the DOM comes in more application-oriented interfaces: the HTML interface is orientated towards visual representation and, for example, includes access to style sheets and events. The
XML interface focuses on higher-level data-centric processing, which means that documents are primarily considered data structures, and enables traversals on the parse tree; furthermore, it defines an event model and support for namespaces along with other useful extensions.

To support the evolution of XML processing, the W3C wants the DOM to evolve on several levels. The first level is the core as described in the last paragraph. Its main objective is to provide support for document navigation and manipulation. The second level additionally features a style sheet object model, and defines functionality for manipulating the style information which a user may provide to annotate a document or, for example, for the visual representation of the document. It also enables traversals of the parse tree, defines an event model and provides access to namespace information. The third level will address persistence issues like document loading and saving. As well, it aims at validation of content models as defined in DTDs and XML schemas. In addition, it will also address document views and formatting, key events and event groups. Future levels may concentrate on interactive features like interaction with a window system such as X-Windows and user input. There are also plans to include the query language XQuery, and to address multi-threading, synchronization, security, and repository storage. Note that the DOM does not specify physical data structures; instead, an implementor is expected to use it as an interface to proprietary data structures.

To give an impression of the flavour of the DOM specification, Figure 2.7 shows a small fraction of the Java version of the XML part of the DOM interface as it is found in [W3C98a]. Note that the interface Document contains XML-specific functions, e.g., one for the creation of attributes of syntax nodes, but that the arguments of the function are generic, i.e., strings. It is possible to create a generic attribute but not to create a specialised key attribute as in the bibliography example. Schema languages can be used to create more application-specific interfaces.

### 2.3.3 Query Languages

Compared to the lower-level APIs of the previous section, XML query languages are another way to navigate through documents. They are usually declarative, i.e., users specify what information they want to extract from documents rather than how it should be extracted algorithmically, which is basically what the DOM is supposed to support. Some query languages were mentioned in the Introduction. For example, if a user wants to have a list of all people who are listed as authors of articles in our example bibliography, he could write in the query language XPath:

```xml
document("bibl.xml"]/bibliography/article/author/text() 
```

This query reads as follows: In the document `bibl.xml` find and return all the text strings at the end of the tag sequence bibliography, article, author. Tag sequences are also called path expressions. The expected result is the list `Ben Bit`, `Bob Byte`, `Ken Key`. If the user is interested in all authors, no matter in what kind of publication they are listed, he can write:

```xml
document("bibliography.xml"]/author/text() 
```

The two simple queries show the basic way to access information in XML documents: the element hierarchies play a role similar to relations in query languages like SQL [Ame86]. Path expressions are used to specify which parts of a document the query should return and store in the result variable. The second example demonstrates
how wildcards ('//') can be used to denote any sequence of tags rather than a specific element. Query languages and their execution on large collections of documents are dealt with in more detail in later chapters.

2.4 XML and Databases

The explanations of the previous section were given in a document context. How do they transfer to the database world? What are the primitive operations a data management system should provide? And on what logical data model? [Mur01] argues that 

*hedges*, ordered sequences of ordered trees, are the appropriate atomic elements while [JLSK01] plea for trees being the appropriate abstraction. In the rest of this thesis the author will present his vision of XML and data management.

There is a basic underlying assumption in nearly all work on mapping and querying XML in Databases. It is assumed that the tags and attributes of XML documents are indeed content models, i.e., that they carry semantics that are close to the natural language equivalents of the tag names. Only then is it possible to equate entities as in E/R models [Tha00] and elements or attributes as in XML. XML and the surrounding framework like DTDs, XML Schema, and the query languages have no way to enforce this assumption; nevertheless, if it is neglected, results may become unintuitive and systems hard to maintain.

2.5 Conclusion

In its basic form, XML is just syntax: it describes a way to combine content particles into structured documents. In many applications, it is assumed that document designers use the derivations, also called content models, that XML provides similarly to semantic modelling processes like E/R modelling. With more powerful modelling features for content models like DTDs or schemas, the user can get closer to the systematic rigidity of data models as deployed in database management systems. In exchange for these efforts, he gets more control over the semantic integrity of the data that are used in a system; additionally, some semantic integrity checking can be automated.

2.6 Bibliographical Remarks

Predecessors of XML are SGML, Bracketed Grammars, where each derivation rule application step is marked ('tagged') [GH67] with unique symbols, and Regular Right Part Grammars [LaL77] which allowed for the introduction of content models. The latter two were combined in SGML [ISO86, Gol90] and, in more strict form, into XML.

[LCO0] give an overview of six XML schema languages in a database setting. [Mur00] presents an excellent formalization of different XML schema languages. Four classes of tree languages are introduced to capture different semantic principles and make their constraints expressible: *local*, *single-type*, *restrained competition*, *regular*. [Dod01] discusses when to use what kind of database product.