Chapter 2
Fundamentals

Abstract The primary purpose of the chapter is to introduce the basics for reading and writing text with XML markup. The logical structure of an XML document includes primarily nested elements, some of which have associated attributes. A document type definition (DTD) can be included to constrain the contents and structure of the document. The concepts of *well-formedness* and *validity* of documents are defined. Two alternative constraining mechanisms, XML Schema and RELAX NG, are introduced and compared to DTDs. Finally, the two standard processing models for XML, one based on streams and one based on trees, are introduced. Although not all details of XML are covered, the chapter provides some literacy with respect to XML specifications, so that the complete language can be learned as necessary.

Keywords DTD • Formal grammars • Logical structure • Namespaces • Physical structure • Processing models • RELAX NG • Schema languages • XML documents • XSD

Chapter 1 introduced markup and some of the basic ideas and uses of XML. This chapter describes XML in more detail and how it is processed by computers. Like HTML, the unit of communication when using XML is a document.

> The abbreviation XML comes from the name *Extensible Markup Language*. XML was developed initially for representing information in the context of the Internet. When using XML, software applications store and exchange data in the form of documents. Within those documents, structural elements are named and marked up in a systematic way to facilitate applications' processing of the elements. XML is a restricted form of SGML,[1] an older markup language.

The rules constraining all XML documents are defined in XML specifications published by the World Wide Web Consortium (W3C) as W3C Recommendations [23].

[1] SGML, the Standard Generalized Markup Language, was accepted as ISO standard 8879 in 1986 [16] and later augmented by supplements [17].

A. Salminen and F. Tompa, *Communicating with XML*,
DOI 10.1007/978-1-4614-0992-2_2, © Springer Science+Business Media, LLC 2011

The first W3C Recommendation for XML was published in 1998 [5]. Since then, four new editions of version 1.0 have been published, in 2000, 2004, 2006, and 2008. These four specifications do not define a new language version but provide corrections for errors discovered after publication of the previous edition. A new version 1.1 of XML was published in February 2004 [6], with a second edition in 2006.

The history leading to the development of XML, as well as the history of XML itself, is summarized in Appendix B. The XML development at W3C is closely related to both software development for processing XML and applications development activities where XML-related representation and communication languages are developed around the world for various communities and application areas. We describe the development process for W3C specifications in Sect. 3.1.

We start this chapter by first introducing the way XML and XML-based languages utilize formal grammar rules to define constraints for languages. In Sect. 2.2 we introduce the processing context of XML documents, which is needed to understand the role of various XML document components in XML communication. Then in Sect. 2.3 we give an overview of the data structure called *XML document*. The definition capabilities included in XML documents are described in Sect. 2.4. Finally, Sect. 2.5 introduces two alternative processing models for XML documents: stream processing and tree processing.

2.1 Formal Grammars

Natural and formal languages are both important in Web communications. Natural languages are used by humans to communicate among themselves; no technology need be involved unless the humans are separated from each other in space or time. For a particular natural language, there may be written rules for building valid sentences in the language, but the rules do not cover the full variety of expressions used by people. Furthermore, people can use a natural language long before they learn to articulate its rules.

XML documents very often include parts written in a natural language, but XML itself and the XML-based markup languages are formal languages, defined by formal grammars. A *formal language* is a set of character strings for which the characters are taken from a given alphabet and concatenated into strings according to exact rules. The language consisting of all positive numbers (that is, strings of digits, such as 301992, 7, and 4124) is an example of a formal language: we can define exact rules for testing whether or not a string belongs to the language. The following are other examples of formal languages:

- SGML, HTML, XML, and other markup languages deriving from SGML
- C, Java, C++, and other programming languages
- Algebraic expressions
- Roman numerals
- Social security identifiers
- Genetic code

The examples show that formal languages are usually developed by a group of people for a specific purpose, such as for computer programming or to represent information in computers. Such languages are developed to facilitate communication with a computer and between computers. Formal languages however seem to evolve also without human activities. The genetic code is an example that exists in nature and is documented as a formal language by humans.

The syntax of formal languages is often specified by a *formal grammar*. A formal grammar (or simply *grammar*) describes the strings of the language by a set of *production rules* (also simply called *rules* or *productions*). *Extended Backus–Naur Form* or *EBNF form* is a notation commonly used to describe such rules. EBNF is used in the XML specifications to describe the acceptable expressions that make up XML. The notation is introduced in the XML specification and in Appendix C of this book.

Each rule in the XML grammar describes one named part, using the form

```
symbol  ::=  expression
```

The symbol on the left is the name of the part, and the expression on the right describes the structure of the part. As examples, some productions of the XML 1.0 specification are shown here, where the number in brackets refers to the number of the production in the full specification.

```
[1]    document    ::= prolog element Misc*
[3]    S           ::= (#x20 | #x9 | #xD | #xA)+
[22]   prolog      ::= XMLDecl? Misc* (doctypedecl Misc*)?
[27]   Misc        ::= Comment | PI | S
[39]   element     ::= EmptyElementTag | Stag content Etag
                           [WFC: Element Type Match]
                           [VC: Element Valid]
```

The first production defines the structure of an XML document. The order of the components on the right side is significant: the notation A B means that A comes before B. Thus, a document always contains a prolog followed by an element. After the element there are zero or more occurrences of parts called Misc, the potential omission or repetition being indicated by the *metasymbol* * (*asterisk*).

Production 3 defines the white space used in XML markup between structural components. It consists of space characters (#x20), tabs (#x9), carriage returns (#xD), or line feeds (#xA). Symbols of the form #xN in the XML syntactic notation refer to a particular Unicode character having code value corresponding to the hexadecimal integer N. Alternatives are separated in the production by the metasymbol | (*pipe*). The parentheses are used to group one part of the rule, and the metasymbol + (*plus*) following the ending parenthesis indicates that whatever is represented by the group can be repeated one or more times. Thus in XML, whitespace can be any string of one or more characters, each chosen from any of the four alternatives. Note that unlike the constituents of Production 1, the order of the individual characters comprising white space is not constrained.

Production rule 22 shows that a prolog may begin with an XML declaration (XMLDecl), where the metasymbol ? (*question mark* or *query*) shows that this part is not mandatory. The following Misc may occur zero or more times (indicated by the metasymbol *). At the end of the prolog there can be a document type declaration (doctypedecl) followed by zero or more Misc occurrences, again the ? indicating that this group is optional. Since none of the three components of a prolog are mandatory, a prolog can be an empty string (that is, a string consisting of no characters at all).

The part called Misc is defined in production 27, which specifies one of three alternatives: Misc is either a comment, a processing instruction (PI), or white space (S).

Production 39 defines an element. According to the production, an element is either an empty-element tag or it consists of a start-tag, content, and end-tag, in that order. Production 39 shows also the notation for associating two special kinds of constraints with productions: *well-formedness constraints* of the form [WFC: ...] and *validity constraints* of the form [VC: ...]. This production is associated with a well-formedness constraint called Element Type Match (which specifies that an element's end-tag must match the element type in the start-tag) and a validity constraint called Element Valid (which specifies the conditions for the validity of an element). The concepts well-formed and valid are introduced in the next section.

The document type definition (DTD) mechanism of XML is a tool for describing the production rules for a particular XML language. The DTD, together with the rules expressed in the XML specification, defines a formal language. Some of the notations used in the production rules of the XML specification are also used in DTDs to define the structure of elements. In defining element structures, for example, the metasymbols (,), ?, |, *, and + are used to refer to grouping, optionality, alternatives, and iteration, just as they are in the XML rules. This is elaborated in Sect. 2.4.2.

2.2 Processors and Applications

The XML specifications from W3C describe not only the structure of XML documents but also some essential aspects of the behavior of computer programs that process XML documents. There are two types of software modules mentioned in the specifications. A software module called an *XML processor* is used to read XML documents. The XML processor is not an independent software module but always works with another software module called an *application*. The XML processor provides the application access to the content and structure of documents (see Fig. 2.1). Although functionally distinct, often the XML processor is embedded as part of the application.

As explained in Chap. 1, an XML document contains marked up text and possibly other forms of data linked to the text by entity references. The XML processor reads the marked up text, separates the markup from other content, and checks that the text conforms to the rules defined for all XML documents in the XML specifications. Those rules are called *well-formedness rules* and documents fulfilling the

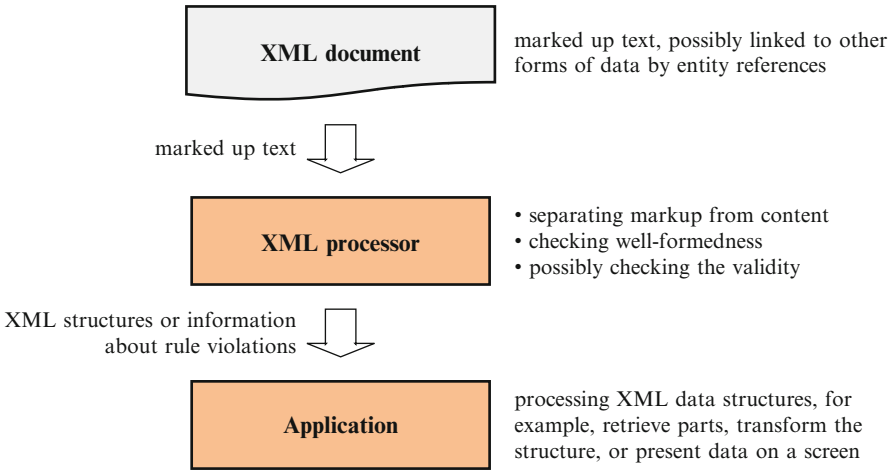

Fig. 2.1 The processing context of an XML document

rules are called *well-formed*. The checking is applied only to the marked up text, not the non-textual content linked to the marked up text by entity references nor the textual content indicated explicitly to be omitted from processing.

If an XML document has an associated DTD, then an XML processor may ensure not only that the document satisfies the well-formedness rules, but also that it satisfies the rules expressed in the DTD. An XML processor capable of performing this kind of testing is called *validating*, and the documents passing the tests are called *valid*. If an XML processor is only capable of testing the well-formedness of the input data, it is called *non-validating*.

Since an XML processor ensures that the input text follows defined syntactic rules, the processor is often called a *parser*. In computing environments, a parser is created for a particular language L, having particular grammar rules. The minimum functionality of the parser of any language is to inform the user (whether a human being or a software module) whether or not the input text belongs to the language L. Thus the parser in any XML processor tests whether the input belongs to the generic XML language. If the input is associated with a schema for a document type T, then the parser of a validating XML processor further tests whether the input belongs to the particular XML language called T.

Example 2.1 To give a more concrete idea about the tasks of an XML processor, we present a small example. Let the input be a file consisting of three text lines as follows:

```
<?xml version="1.0"?>
<!DOCTYPE plain_text [ <!ELEMENT plain_text (#PCDATA)> ]>
<plain_text>Today's weather is truly exceptional.</plain_text>
```

The first line shows the XML version for the markup used, the second line indicates the DTD, and the third line contains an element called plain_text. If the text is given as an input to a non-validating XML processor, the processor should check that the markup in the text follows the rules given for well-formed documents in the XML 1.0 specification. The processor should also check the syntactic correctness of the DTD. However, no comparison between the DTD and the rest of the text is made. On the other hand, if the same input is given to a validating XML processor, it does all the same tasks, but it is also able to compare the DTD to the element to check the validity constraints given in the specification of XML 1.0. However, for any XML processor the content of the only element is merely a string of characters; no XML processor is capable of testing whether the content of the element is proper English (although an XML application outside the XML processor might implement such functionality).

The task of an XML processor is to identify individual units of content, as well as the relationships between those units, and forward the information about them to an application. The application in an XML processing environment is any software module able to deal with XML data. Thus the application is the real consumer of XML data; the XML processor only checks the input and forwards the structural components to be handled by the application. For example, the application might connect to a database system to store the structural components into its internal structures. On the other hand, it might be capable of manipulating natural language text instead, and so it might be able to check that the content of the element in Example 2.1 is proper English.

Similarly to the parsers of programming languages, the XML processor informs its user (the application) about any violations of syntax rules in the text. The XML specifications define two kinds of violations of rules: *errors* and *fatal errors*. An XML processor may detect and report an error, and may recover from it, but not all XML processors need to detect errors. Fatal errors are kinds of violations that all conforming XML processors *must* detect and report to the application. Applications may be designed to handle data where the XML processor has found errors, but XML-conforming applications may not attempt to work normally when they are informed of fatal errors.[2] In the Web environment many authors of HTML documents are familiar with the flexibility of HTML browsers, which often represent documents that violate HTML rules in a manner in which the violations remain unnoticed. Typically, no information about the violations is given to the human reader of Web pages. This can also happen with XML documents for errors, but not for fatal errors. Thus an application's accommodation of XML data does not guarantee validity of a document, but well-formedness is assured.

The small example given above is intended to clarify the distribution of labor between an XML processor and application. In the following section we take a closer look at the components of XML documents.

[2] "Once a fatal error is detected, however, the processor *MUST NOT* continue normal processing (i.e., it *MUST NOT* continue to pass character data and information about the document's logical structure to the application in the normal way)." "This innocent-looking definition embodies one of the most important and unprecedented aspects of XML: 'Draconian' error-handling." [3]

2.3 XML Documents

Every XML document has a logical structure and a physical structure. Figure 2.1 shows an XML document as if it were stored in a single file. However, a document may consist of several physical files. In this section we first discuss the logical structure, then the physical structure, and finally the character encoding for XML documents.

2.3.1 Logical Structure

Effective communications among humans relies on organizing ideas into meaningful information structures. XML offers the means to present such communications in a form in which the structures and structural units are processable by software applications. The organization is expressed in the logical structure of documents, and the most important components of that structure are *elements*.

2.3.1.1 Elements and Nested Structures

Elements, like all other components of the logical structure, are indicated by explicit markup. As discussed in Chap. 1 and shown in our previous examples, elements in XML begin with a *start-tag* of the form < ... > and end with an *end-tag* of the form </... >. Both the start-tag and the end-tag include the name of the element.

XML does not set many restrictions for element names. They must start with a letter or an underscore (_) and can include arbitrarily many alphanumeric characters as well as periods, hyphens, underscores, diacritics, and various other typographic marks; most notably they cannot include white space characters. For example, if we want to label the string *1654* with the name *year*, we can present it as an XML element:

```
<year>1654</year>
```

The text between the tags is called the *content* of the element. Within a document, all elements having the same name belong to the same *element type*, and most applications are written such that all elements of one type are similarly processed. For example, a banking application could include processing code to handle all elements named deposit.

An element without content can be written without the end-tag, using a special kind of *empty-element tag* of the form < .../>. For example, the empty-element tag
 could be used to inform a printing application about the need for a carriage return. This is identical to the tagged string
</br> with no intervening characters.

An element's content can consist of plain character data, such as the year element above or the plain_text element in Example 2.1, or it can contain other elements as

child elements. The child elements contained in a common *parent element* are called *sibling elements*. For example, an element showing a date could be written in the form

```
<date><day>11</day><month>12</month><year>1654</year></date>
```

This date element contains three child elements, named day, month and year, which are therefore siblings. The level of nesting is unlimited: any child element can itself have nested child elements. The following example shows three-level nesting of elements.

Example 2.2 The following XML document has two rhymes, one written in Finnish and the other in English. The outermost element is called rhymecollection (note that the name cannot include white space characters), and it contains two rhyme elements as its children. Each of the rhyme elements, in turn, consists of two line elements.

```
<?xml version="1.0"?>
<rhymecollection>
<rhyme>
<line>Ole aina iloinen</line>
<line>niin kuin pikku varpunen</line>
</rhyme>
<rhyme>
<line>See, see! What shall I see?</line>
<line>A horse's head where his tail should be</line>
</rhyme>
</rhymecollection>
```

An application showing tagged text to the user may highlight the nesting of elements through indentation. For example, if the text above is stored in a Microsoft environment as a file named with extension .xml, and the file is opened by Internet Explorer 6.0, the document is shown in the pretty-printed form:

```
<?xml version="1.0" ?>
-<rhymecollection>
  - <rhyme>
      <line>Ole aina iloinen</line>
      <line>niin kuin pikku varpunen</line>
    </rhyme>
  - <rhyme>
      <line>See, see! What shall I see?</line>
      <line>A horse's head where his tail should be</line>
    </rhyme>
</rhymecollection>
```

The minus character (−) preceding a start-tag indicates that the element has child elements and the child elements are presented on the screen. By clicking the

minus character, the child elements and the matching end-tag will be hidden and the minus character will be replaced by plus character (+). For example, after clicking the minus characters preceding the two rhyme elements, the document would be shown as

```
<?xml version="1.0" ?>
−<rhymecollection>
 +    <rhyme>
 +    <rhyme>
</rhymecollection>
```

All well-formed documents in XML have some restrictions on their nested structures, including the requirements for a single root element and proper nesting of elements. Thus, in the nested structure there is always exactly one outermost element, called the *root element* (or *document element*), and all non-root elements are fully contained in some other element. Thus, for example, in representing a collection of rhymes in an XML document, the rhymes have to be nested inside a common root element as was done in Example 2.2; there cannot be two elements at the outermost level of the nesting structure. *Proper nesting* means that if the start-tag of an element is part of the content of another element, the end-tag must also be part of the content of that same element. For example, markup such as

```
<date><day>11<month></day>12</month><year>1654</year><date>
```

is not correct in any XML document since elements day and month are not properly nested.

2.3.1.2 Unparsed Character Data

Text in XML documents consists of intermingled markup and character data. All text that is not defined as markup is character data. For cases where characters denoting markup should be included in the character data, a mechanism called a CDATA section is available. A CDATA section begins with markup <![CDATA[and ends with the markup]]>. All characters within these delimiters are regarded as character data, not markup. For example, to include the string

```
<lastname>Pirhonen</lastname>
```

as character data in some XML document, the following CDATA section would be used:

```
<![CDATA[<lastname>Pirhonen</lastname>]]>
```

This assures that none of the enclosed text is interpreted as markup.

2.3.1.3 Attributes

In the rhyme collection of Example 2.2, English and Finnish were the languages used in the rhymes. In this kind of situation, it might be useful to inform the application about the language in which a rhyme is written. This kind of extra information, or *metadata*, can be attached to elements by using *attribute specifications*. Attribute specifications can be added to the start-tag of an element, following some white space after the element name. An attribute specification gives the *name* of an attribute and a character string as the *value* of the attribute. For example, we can specify that a rhyme is in Finnish as follows:

```
<rhyme lang="FI"> ... </rhyme>
```

Notice that the end-tag does not repeat the attribute specification.

As a second example, assume the lastname elements in a document are used to give the current surname of a person. A former surname might be given by using the attribute earlier:

```
<lastname earlier="Rantanen">Korhonen</lastname>
```

The value of the attribute earlier for the element is Rantanen.

This example shows that there are two different techniques for providing a piece of data in XML elements: as element contents and as attribute values. In both cases a name can be attached to the datum. In the example above, the current surname of a person is expressed as the content of the lastname element and the former surname is given as the value of the attribute named earlier. It is also possible to give both names as child elements of the lastname element:

```
<lastname><current>Korhonen</current><earlier>Rantanen></earlier></lastname>
```

In this version, the current surname is given first in a child element named current and then the earlier name is given in a child element named earlier. The names could instead be given in the reverse order:

```
<lastname><earlier>Rantanen></earlier><current>Korhonen</current></lastname>
```

Yet another alternative is to give both names as attribute values of an empty element:

```
<lastname earlier="Rantanen" current="Korhonen"/>
```

or

```
<lastname current="Korhonen" earlier="Rantanen"/>
```

All of these alternatives provide information about the current and former surnames of a person. When the names are given as child elements of the same element, the XML processor informs the application about the order of the names. On the other hand, the order of attributes is insignificant. Therefore, from the point of view of an application, the last two alternatives are indistinguishable, whereas the others are all different.

2.3.1.4 Comments and Processing Instructions

As well as elements, which are the core components of the logical structure, other components of the structure are declarations, comments, and processing instructions. The XML declaration shown in Example 2.1 always starts an XML document. In XML version 1.0 the XML declaration is not mandatory, but it is in version 1.1. Nevertheless it is advisable to use it in all XML documents. Among the other declarations, the most essential are the markup declarations, which are described in Sect. 2.4.

Comments can be written to provide some extra information to the human reader of the marked up text. A comment begins with the character string '<!--' and ends with the string '-->', for example,

```
<!-- This is a comment -->
```

An XML processor may, but need not, make the text of the comment accessible to the application. Thus if comments are to be used by an application, it is important to ensure that the XML processor preserves them; however, to ensure robustness, applications are better designed if they treat comments as being completely optional.

Instead of using comments to pass information to an application, XML provides *processing instructions* to allow documents to contain instructions for applications. A processing instruction begins with the character pair '<?' and ends with the pair '?>'. The instruction is passed to the application and identified by a target name at the start of the instruction. The rest of the instruction is any character string meaningful to the application. The target name is intended to identify the application component to which it is directed. An example of the use of processing instructions is to provide information about an associated style sheet to some application rendering the document, as recommended by W3C [8]. For this purpose, the target name in the processing instructions is xml-stylesheet and additional information is provided by pseudo-attributes such as shown in the following example:

```
<?xml-stylesheet href="ownstyle.css" type="text/css"?>
```

The string after the target name xml-stylesheet looks like two attribute specifications, such as those that would be written inside the start-tag of an element. They are called pseudo-attributes because they are used to provide information to the application in the same way that attribute specifications are used, but they appear within a processing instruction rather as part of a start-tag. The example provides the application with information about an external CSS style sheet, in the same way as a style sheet association is given in HTML:

```
<LINK href="mystyle.css" rel="stylesheet" type="text/css">
```

2.3.1.5 Namespaces

A goal of the work at W3C is to support the reuse of XML structures once they are developed, especially if there is software available to support the processing of those structures. For example, structures relevant to banking (including withdrawals and deposits) may need to be handled by banking software and those relevant to mining (including the analysis of deposits and soil formations) may be handled by mining software. When reusing elements and attributes defined in previous environments, it is important that the XML processor can identify the context for each name and that document developers avoid name collisions. For example, element types named Title have been introduced in many different environments for different purposes. In one context the Title elements may be used for publication titles consisting of characters only, in another the Title elements may be used for property titles and contain child elements, and in a third Title elements may be used for titles of persons. If a developer wishes to use several or all of these element types in a single document, it is important that unintentionally duplicated names of elements or attributes can be distinguished.

As a more concrete example, let us consider the XHTML language, which describes the structures available in HTML, but is constrained so that XHTML documents are also XML documents. The structures and names defined in XHTML are widely known, and there is plenty of software able to process XHTML structures. It might be that some documents combine XHTML structures with structures from other XML-based languages, such as XQuery or MathML; in this situation, the processor should forward to the application information about the context within which to handle each structure. By following this approach, XHTML applications can manipulate the XHTML structures and ignore the others, if they wish to do so.

In order to support modularity of specifications and reuse of element definitions without name collisions, W3C has developed a method to associate an environment identifier with several element and attribute names so that the context can be recognized when they are used. Since the idea was developed after the original design of XML, the method is not described in the XML specification but in a separate *XML Names* specification [4].

A set of element and attribute names together with their identifier is called an *XML namespace*. The original method of identification was to use a *Uniform Resource Identifier* (*URI*) reference, which can be a URL (Uniform Resource Locator) that may be familiar from HTML links, or a URN (Uniform Resource Name). For example, the URI used to identify the elements and attributes defined in XHTML is http://www.w3.org/1999/xhtml, as indicated in the XHTML specification. In the new version of the XML Names specification, the identification mechanism was extended to *Internationalized Resource Identifier* (*IRI*) references [13]. Whereas a URI is a string of characters chosen from a subset of US-ASCII characters, an IRI extends URIs to a wider set of characters so that they can be used in the context of various natural languages.

A *namespace label*, which follows the rules for an element name, is bound to an IRI by a *namespace declaration* in the start-tag of the element where the namespace

is introduced. For example, in the following element start-tag the label xhtml is associated with the IRI representing the XHTML namespace:

```
<report  xmlns:xhtml = "http://www.w3.org/1999/xhtml">
```

Any names from this namespace are then referenced by using *qualified names* consisting of the label as a prefix and a name in the namespace, separated by a colon. Continuing with the example, after the namespace declaration introducing xhtml, any structural element or attribute from XHTML can be used within the element report by prefixing xhtml: to its name, producing qualified names such as, xhtml:meta or xhtml:h1. Thus, for example, within the report element, one might find

```
<xhtml:a  xhtml:href = "example/figure">our results</xhtml:a>
```

Note that a namespace name can use any label, as long as it is distinct from other namespace labels. Therefore, instead of the label xhtml for the XHTML namespace we could define

```
<report  xmlns:abba = "http://www.w3.org/1999/xhtml">
```

and then refer to the href attribute of the XHTML language as abba:href.

The XML Names specification does not constrain how the names within a namespace are defined. Often a namespace consists of all element and attribute names introduced in a document type definition or other kind of schema. However it is possible to specify the set of names included in a namespace by other means as well. In any case, to be able to refer to the names from a namespace requires the identification of the namespace by an IRI. Note that even if the namespace identifier is syntactically a URL, it does not refer to any location on the Web; rather it is simply a unique identifier. For example, if people in the sales department of MyCorp agree to use the URL http://sales.mycorp.com to identify a set of element and attribute names, this URL can be used for specifying the namespace, whether or not there is some data at that Web address.

2.3.2 Physical Structure

Each XML document has a physical structure as well as a logical one. The physical structure facilitates features not expressible by the logical structure alone. For example, the physical structure allows the inclusion of non-textual data in a document, even though the logical structure deals with marked up textual data only. Thus an XML document can also be a multimedia document. Building software to deal with the logical structure alone misses an important aspect of XML processing.

The physical structure of an XML document consists of units called *entities*. Each document includes a designated text entity called the *document entity* or *root entity*.

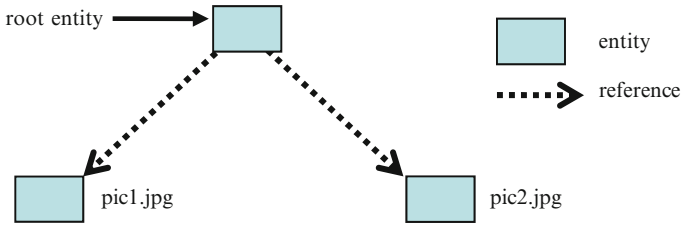

Fig. 2.2 A physical structure with three entities

All entities referred to directly or indirectly from the root entity are regarded as parts of the physical structure of the document. A common type of an entity is a non-textual file referenced from a text entity, and non-textual entities may include data in any format. Figure 2.2 shows the physical structure of a document with a root entity and two jpg images as non-textual entities.

2.3.2.1 Entity Types

Entities can be categorized according to three characteristics. The first indicates whether or not XML markup is to be interpreted: an entity is either *parsed* or *unparsed*. The second characteristic indicates where the content of an entity is given: an entity can be *internal*, in which case the content is given in the entity declaration, or *external* in which case the content is given as a separate physical object. The third characteristic indicates the use of entities: an entity is either a *general* entity, in which case it is used within the elements of the document, or a *parameter* entity used within the DTD. These three characteristics of entity types are further described as follows.

Parsed – unparsed. The content of a parsed entity consists of marked-up text intended to be analyzed by the XML processor. The root entity is always a parsed entity. An unparsed entity is a resource whose content may, or may not, be text; and if it is text, it may or not include XML markup. Regardless of content, however, an unparsed entity is not intended to be analyzed by the XML processor, and XML places no constraints on the content of unparsed entities. In Fig. 2.2 the document includes the tagged text of the document as a parsed (root) entity and the two figures as unparsed entities.

Each unparsed entity has an associated *notation*, identified by name. The notation informs the application of the data format of the entity so that the content can be managed appropriately.

The tagged text of a document need not be all stored at the root; it may be divided into two or more parsed entities. The ways text can be split into parsed entities is regulated by the rules defined for *well-formed parsed entities* in the XML specification, and in a well-formed document, all parsed entities must be well-formed. The well-formedness rules are defined so that the logical and physical structures in an XML document are properly nested. For example, both the start and end of an element must be within the same (physical) entity.

Example 2.3 In Example 2.2 a collection of rhymes was presented as an XML document stored in one entity. The rhyme collection can be divided into two or more parsed entities, but all of them have to be well-formed. For example, an entity with the following content is not acceptable because it contains the end-tag of the element rhymecollection but not its start-tag:

```
<rhyme>
<line>See, see! What shall I see?</line>
<line>A horse's head where his tail should be</line>
</rhyme>
</rhymecollection>
```

On the other hand, the text

```
<rhyme>
<line>See, see! What shall I see?</line>
<line>A horse's head where his tail should be</line>
</rhyme>
```

or the text

```
See, see! What shall I see?
```

or even

```
hat shall I se
```

could appear as the content of a well-formed entity.

External – internal. Memory units managed as separate files in XML processing environments represent external entities. An alternative type of an entity is an internal entity, a named piece of text contained within another entity and intended to be analyzed by an XML processor. The text piece is called the value of the entity. An internal entity is always a parsed entity. This kind of entity is used, for example, to avoid repetitive writing of long or otherwise complex pieces of text. For example, if we assign the name UJ to the string

```
University of Jyväskylä
```

we can include the longer text value in a document by referring to UJ. References to named, parsed entities are called *entity references*. In processing the XML document, the XML processor replaces the name by its value before transmitting data to the application. The syntax of entity references will be given below in Sect. 2.4.5.

Parameter – general. The separation of entities into general and parameter entities is based on the context of their use in a document. A parameter entity is for use within a DTD, whereas a general entity is for use within the element content of a document. A parameter entity is always a parsed entity.

Table 2.1 The five possible combinations of entity types

Parsed	Internal	Parameter
Parsed	Internal	General
Parsed	External	Parameter
Parsed	External	General
Unparsed	External	General

To conclude the characterization of entity types, let us consider the possible combinations of characteristics. Even though there are two possible settings for each of the three characteristics, all combinations are not possible. An unparsed entity cannot be internal, nor can it be a parameter entity. On the other hand, parsed entities can be either internal or external and either general or parameter entities. Therefore there are five possible combinations for characteristics of entity types, as summarized in Table 2.1.

2.3.2.2 Motivations for the Use of Entities

Casual authoring of XML documents can be done without knowing much about entities. In professional use of XML, however, entities play an important role.

There are several reasons why it is often important to divide an XML document physically into multiple pieces. We have already mentioned one: when *non-textual data* is to be included in XML documents, external unparsed (general) entities are needed. However, parsed entities can also provide support to document creators.

The possibility to name a piece of text and to refer to it by an entity reference, instead of writing it in full, is a valuable mechanism to ensure consistency when that text must be repeated many times. Just as the namespace mechanism facilitates reuse of element and attribute definitions originating from different sources, the entity mechanism facilitates reuse of text fragments, either in the element content or in the DTD. For example, a title appearing several times in a document can be defined as a general entity. Similarly a piece of a definition intended to be used in the DTD in several places can be defined as a parameter entity. In its simplest form, the repeated text is a single character, more particularly a special character not directly accessible from the keyboard. However, the entity can also be an arbitrarily large fragment of boilerplate.

Referring to a piece of text by its name, instead of using the text itself, often simplifies writing, but more importantly it supports *consistent* writing. This is especially important when boilerplate is required and in environments where several content authors coordinate in writing texts. For example, in a text about W3C technical specifications repeatedly mentioning the terms "W3C Recommendation", "W3C Proposed Recommendation", "W3C Candidate Recommendation", and "W3C Working Draft", and in the absence of controls, authors might easily write the terms in slightly different ways. If entity names, such as Rec, PRec, CRec, and WD, are used instead, then the terms for the various kinds of W3C technical specifications will appear in the final documents in a consistent form, in spite of sections being written by distinct content authors.

The naming facility is especially important for *modularity* and *maintainability*. For example, several schema designers may be involved in the design of a large DTD, which evolves through several versions. The work can be partitioned into modules by dividing the DTD into several entities where each of the designers focuses on his or her own module. If a designer makes changes to the definitions in one module, the other designers using the definitions by reference need not make any changes to their modules. Similarly if an organization's legal department updates the wording of a disclaimer, and that disclaimer is stored as the content of a parsed entity and referenced from each place it is required, the new wording will automatically be incorporated in all the organization's documents.

Entities can also be used to provide consistent *semantic information* aimed at human readers. For example, consider a situation where an attribute for a date is used in several element types. If the schema for the class of documents is defined by the DTD mechanism, the capabilities to constrain the attributes' values are very limited. However, the schema designer can provide information about the intended form of the dates by defining a parameter entity, say Date, that includes the following comment:

```
<!--a date given by eight digits in the form DDMMYYYY, for example, 24022005 -->
```

Such a comment might help document authors to write dates consistently. If consistent date forms are used, an application could be implemented to recognize the name Date as an attribute type and safely assume that the attribute values will be eight digit numbers representing DDMMYYYY.

2.3.3 Character Encoding

Text in XML documents is encoded using the Unicode character set. Unicode is intended to serve as a character set for representing textual data written in any natural language of the world; it is even independent of the writing direction of the language.

The set of characters available in Unicode is huge, and therefore mechanisms are needed to be able to express a particular character without the need to use a symbol representing that character in a particular natural language. For example, this is especially important in situations where a character is not directly accessible from the keyboard or other available input device. There are two ways to refer to single characters without using their symbols. First, as we mentioned in Sect. 2.3.2, the value of an entity may be merely a single character; how to reference such an entity will be described in Sect. 2.4.4. Secondly, a single character can be referred to by a *character reference*, without defining an entity for that character. A character reference provides a decimal or hexadecimal representation of a character's code point in Unicode. The reference begins with the characters &# and ends with a semicolon (;). The letter x after the characters &# signals the use of hexadecimal representation. For example, " or " refers to the quotation mark by specifying code point 34 (hexadecimal 22) in the Unicode character set.

One problem related to character coding is the continuous evolution of natural languages. Since languages evolve, Unicode must also evolve. This evolution was not realized in the specification of version 1.0 for XML, where the characters used in XML markup, like the element and attribute names, were defined to be characters from Unicode 2.0. This has caused limitations in the use of XML since Unicode has subsequently been extended to versions 3.0 and 4.0, and there will be later versions in the coming years. This problem related to Unicode versioning was one of the major reasons for developing the new version 1.1 for XML.

2.4 Declaring and Constraining Structures

Extensibility is an important feature of XML. Like its predecessor SGML, XML is a *metalanguage*, a language for describing other languages. The XML specification does not define the element or attribute names to be used in documents, nor the element structures to be used. The idea behind XML is to agree on a notation for developing special vocabularies and markup languages for particular purposes. Thus anyone can extend the rules provided in the XML specification with his or her own rules to constrain the documents for a particular application area.

2.4.1 DTD and Markup Declarations

Documents based on a common structure (or language) are said to be of the same *document type*, and the structure for a particular language is defined by markup declarations in a *Document Type Definition* (*DTD*). The declarations of a DTD define the accepted element types and attributes, as well as the logical structure of documents of the type. Along with the constraints for logical structure, declarations in the DTD are also used to introduce the entities available for inclusion in documents of the type. The markup declarations can be given either locally, in the root entity of the document in the *document type declaration*, or externally in a separate file, as a separate entity. In the latter case, the address of the file must be provided by the document type declaration. The terms *internal subset* and *external subset* refer to the locally and externally given markup declarations, respectively. The external subset is also an external entity.

In Example 2.1 a document was given with a local DTD:

```
<?xml version="1.0"?>
<!DOCTYPE plain_text [ <!ELEMENT  plain_text (#PCDATA)> ]>
<plain_text> Today's weather is truly exceptional.</plain_text>
```

The document type *declaration* consists of the shadowed line, which contains the DTD (document type *definition*). If the DTD is specified externally, in a separate file,

then the name of the file should be provided in the document type declaration in place of the DTD itself. Thus, an external DTD is attached to a document as follows:

```
<?xml version="1.0"?>
<!DOCTYPE plain_text SYSTEM "mytext.dtd">
<plain_text> Today's weather is truly exceptional.</plain_text>
```

In this example, the system identifier mytext.dtd specifies where to find the DTD for the document.

Four kinds of markup declarations are available for constraining XML documents: element type declarations and attribute list declarations to constrain the logical structure, and entity and notation declarations to constrain the physical structure. Before going into the particular types of declarations, we first give an introductory example to demonstrate how DTDs can be used to constrain documents.

Example 2.4 Example 2.2 above showed a document with two rhymes. The markup vocabulary and structure for the markup used in Example 2.2 can be defined by the following DTD:

```
<!DOCTYPE    rhymecollection [
<!ELEMENT    rhymecollection    (title?, rhyme+)>
<!ELEMENT    title              (#PCDATA)>
<!ELEMENT    rhyme              (line+)>
<!ATTLIST    rhyme
      xml:lang  (fi | en)       #IMPLIED
      author    CDATA           #IMPLIED >
<!ELEMENT    line               (#PCDATA)>          ]>
```

The definition consists of four *element* type declarations and one *attribute list* declaration (ATTLIST). The attribute list declaration introduces two attributes for elements of type rhyme. In the definition, the following constraints are defined for documents of type rhymecollection:

Element names. Only the element names rhymecollection, title, rhyme and line are allowed in the documents.

Attributes. Two attributes can be attached to the rhyme element: xml:lang and author. The data types of the attribute values are defined after the attribute names. In the example, the value of the attribute xml:lang is to be taken from an enumerated list (either the string "fi" or the string "en"), and the value of the attribute author is a character string (CDATA). The repeated keyword #IMPLIED indicates that either or both of the attributes can be given in the start-tag of a rhyme element, but neither of them is mandatory (and any application must infer a missing attribute's value from context).

Structure. The root element of the document is of type rhymecollection. The structural constraints concerning elements of the type are given by the content model (title?, rhyme+) using metasymbols ? and + from the EBNF notation introduced in

Sect. 2.1. It indicates that an element of the type rhymecollection contains one or more rhyme elements, possibly preceded by an element of type title. The content model (line+) defines a rhyme element as a structure consisting of one or more line elements. In the declarations of the element types title and line, the keyword (#PCDATA) indicates that the elements of these two types consist of character data devoid of markup characters.

Although it is well-formed, the tagged text describing two rhymes in Example 2.2 is not a *valid* document of any type, since there is no DTD attached to the text. However, the text meets the element type constraints defined in the DTD given in Example 2.4. Assume that the DTD is accessible at rhymes.dtd. The document of Example 2.2 would then be valid if it includes the following document type declaration after the XML declaration and before the root element:

```
<!DOCTYPE rhymecollection SYSTEM "rhymes.dtd">
```

2.4.2 *Element Type Declarations*

An element type declaration defines a name for a set of elements and constrains the content of those elements. In other words, it specifies what are acceptable element names and what is acceptable between the start-tag and end-tag in each element. The syntax for the element type declaration is given by production rules 45 and 46:

```
[45] elementdecl ::= '<!ELEMENT' S Name S Contentspec S? '>'
                                        [VC: Unique Element Type  Declaration]
[46] contentspec    ::= 'EMPTY' | 'ANY' | Mixed | Children
```

The validity constraint named Unique Element Type Declaration is attached to the element type declaration and specifies that an element name cannot appear in more than one type declaration.

Rule 46 gives four alternatives for constraining an element's content:

- The content must be empty (EMPTY),
- The content is completely unconstrained (ANY),
- The content may directly include child elements and character data (Mixed), or
- The content may directly include child elements only (Children).

If the element type specifies mixed content, then the elements of that type may contain character data optionally interspersed with child elements. Unlike the specification ANY, the types of the child elements may be constrained; unlike Children, the order and the number of occurrences of child elements cannot be constrained in mixed content.

Example 2.5 Suppose that the content of a paragraph should consist of character data interspersed with phrases intended to be rendered in italics or in bold. We could define an element type for such a paragraph where the rendering information is indicated by markup. The following types could be defined for this purpose:

```
<!ELEMENT    paragraph (#PCDATA | italics | bold)*>
<!ELEMENT    italics                (#PCDATA)>
<!ELEMENT    bold                   (#PCDATA)>
```

All three element types are examples of mixed content definitions, but the content of the types italics and bold consists of character data only. A paragraph element may contain characters only, it may consist of arbitrarily many child elements of types italics or bold or both, or it may contain character data interspersed with such child elements, for example:

```
<paragraph>Viljo Revell is the architect of the Toronto City Hall</paragraph>
<paragraph>
  <bold>Viljo Revell</bold>is the architect of the<italics>Toronto City Hall</italics>
</paragraph>
<paragraph>
  <italics>Viljo Revell</italics> is the architect of the <italics>Toronto City Hall</italics>
</paragraph>
<paragraph>
  <bold>Viljo Revell is the architect of the Toronto City Hall</bold>
</paragraph>
```

Metasymbols |, *, (, and) are used in the mixed content definitions with the same semantics as in the XML syntax notation described in Sect. 2.1 above. The list of alternatives must start with #PCDATA, and the order of the child element types has no significance. Since the number of occurrences of child elements cannot be constrained, the symbol * always appears after the alternatives for child elements. An element type definition equivalent to the definition of type paragraph in Example 2.5 is

```
<!ELEMENT    paragraph    (#PCDATA | bold | italics)*>
```

but none of the following describe mixed content in a well-formed XML document:

```
<!ELEMENT        paragraph    (bold | italics | #PCDATA)*>
<!ELEMENT        paragraph    (#PCDATA |bold | italics)>
<!ELEMENT        paragraph    (bold | italics)*>
```

This is an example of the limitations included in the XML specification to simplify the building of XML processors. The incorrect forms of definition could make as much sense as the correct ones, but by constraining the forms allowed in element type definitions, the parsing of XML documents becomes easier and therefore building the software for parsing becomes more straightforward.

As these examples illustrate, the constraining capabilities of mixed content are limited. However, if the content is to include child elements only, without any interspersed character data, the element structure can be constrained by a *content model* where the metasymbols '+', '*', '|', '?', '(', and ')' are available to constrain the element structure. The semantics of the metasymbols is again the same as in the XML syntax notation (see Sect. 2.1). However, whereas no separate symbol is used to indicate concatenation in the XML syntax notation, in content models two successive content particles are separated by a comma (,).

Example 2.6 The following examples show four element type declarations:

```
<!ELEMENT    product      (mfg, model, description, clock?)>
<!ELEMENT    model        (#PCDATA)>
<!ELEMENT    description  (#PCDATA | feature)*>
<!ELEMENT    clock        EMPTY>
```

The first declaration defines an element content model, the second and third specify mixed content, and the fourth constrains the type to be empty. Elements of type product must always begin with an element of type mfg, then an element of type model, and after that an element of type description. At the end there may or may not be an element of type clock. Child elements of type model contain character data only, and description elements may contain character data interleaved with child elements of type feature. If there is a clock child, it will always be empty.

Example 2.7 Document type definition for a phone number:

```
<!DOCTYPE    phone [
<!ELEMENT    phone        (areacode, number)>
<!ELEMENT    areacode     (#PCDATA)>
<!ELEMENT    number       (#PCDATA)>          ]>
```

This grammar defines a language that includes each of the following three sentences:

- <phone><areacode>0146119</areacode><number>2603031</number></phone>
- <phone><areacode>014</areacode><number>university</number></phone>
- <phone><areacode>#5 silly bits</areacode><number>2603031</number></phone>

These examples demonstrate that even if a phone number in XML format is valid with respect to this DTD, it does not necessarily represent the correct form of a phone number as we normally understand it. The DTD mechanism is not always expressive enough to constrain elements' contents to values within the domain intended by the application designer. For this reason W3C has also defined the more powerful XML Schema facility, which is described briefly in Sect. 2.4.6.

2.4.3 Attribute List Declarations

Attributes are used to attach information to elements using name-value pairs. Attributes can be attached to elements in two ways, either explicitly by an attribute specification or by declaring a default value as part of the attribute in the document type definition. In the latter case, the XML processor associates the attribute name and value with the element. Explicit attribute specifications appear either in the start-tag of an element or in the empty-element tag. In a valid document all attributes written in tags must be declared.

An attribute list declaration defines the set of attributes pertaining to a given element type, establishes type constraints for the attributes, and provides default values and constraints on the presence of attributes. The syntax of an attribute list declaration is specified by productions 52 and 53:

```
[52] AttlistDecl ::= '<!ATTLIST' S Name Attdef* S? '>'
[53] AttDef    ::= S Name S AttType S DefaultDecl
```

The first rule states that an attribute list declaration includes a name and zero or more attribute definitions. The name given must refer to the name of an element type. In each of the attribute definitions, we find the name for an attribute, a data type for that attribute, and a default declaration. For example, in the following attribute list declaration an attribute called author is defined for the element type poem; the attribute must be a character string, denoted by the type CDATA:

```
<!ATTLIST        poem       author   CDATA    #REQUIRED >
```

The default declaration in an attribute definition is used to provide information on whether the presence of the attribute is required, and if not, how an XML processor is to react if the declared attribute is not present in an element. The default declaration has four different forms:

- #REQUIRED — the attribute must always be explicitly provided
- #IMPLIED — the attribute may be provided; no default value is given
- AttValue — the attribute may be provided; AttValue (a quoted string) is the default value for the attribute if it is omitted
- #FIXED AttValue — the attribute must always have the default value given by the quoted string AttValue

In the above example, the constraint #REQUIRED indicates that the attribute author must always be explicitly included in poem elements and no default value is given to the attribute. Thus in a valid document the start-tag <poem author="Murasaki Shikibu"> is correct but the stand-alone tag <poem> is not. On the other hand, if the attribute author is instead defined for the element type poem as follows

```
<!ATTLIST        poem       author   CDATA    #IMPLIED >
```

then both of the start-tags <poem author="Murasaki Shikibu"> and <poem> are accepted in a valid document. Similarly, both are valid if a default value is given, as in the following declaration:

```
<!ATTLIST        poem      author   CDATA   "Ono no Takamura ">
```

In this case it is possible to provide the attribute explicitly in the element, as above, but if it is not provided then the XML processor attaches the attribute to the element with the default value. The symbol #FIXED should be specified in the declaration if the default must always be used as the value of the attribute.

While the character data in the content of elements cannot be constrained (#PCDATA allows any characters to be included), attribute values can be constrained to be within any of several data type families: a string type, several tokenized types, and enumerated types. The *string type* is expressed in the declaration by the symbol CDATA as shown in the above examples. *Tokenized types* are used in cases where the attribute value can be constrained into a certain kind of token or list of tokens. There are seven different tokenized types: ENTITY, ENTITIES, NMTOKEN, NMTOKENS, ID, IDREF, and IDREFS. The plural forms refer to lists of tokens separated by white space.

The types ENTITY and ENTITIES are used to specify that the value of the attribute must be a name of an unparsed entity or a list of such names, respectively.

The types NMTOKEN and NMTOKENS are used to specify values that are made up of name characters only. For example, an attribute phase of type NMTOKEN could be defined for an element type report as follows:

```
<!ATTLIST       report     phase   NMTOKEN   #IMPLIED >
```

The start-tag <report phase="draft"> would then be correct while the tag <report phase="preliminary draft"> would not be accepted since a space is not accepted as a name character. Both XML versions 1.0 and 1.1 define constraints for the first character accepted in a name. In version 1.0, only a letter, underscore (_), or colon (:) is accepted as the first character of a name; version 1.1 is not as restrictive, but it also does not accept any of the digits 0–9 at the start of a name.

Types ID, IDREF, and IDREFS allow users to associate unique names with elements in XML documents, and reference those names from other elements. The values of type ID must be tokens accepted as names, and in a valid document every ID value must be unique across all elements that bear any attribute of the type; that is, a name must not appear as the value for an attribute of type ID more than once in an XML document. For example, if attribute article_number has been defined to be of type ID for the element type article and the value A123 appears as the value for article_number on some article element, then it cannot appear as the value of *any* other ID type attribute for another article nor for any other element type.

Each name appearing in the value of attribute types IDREF or IDREFS must appear in the same document as the value of some ID type attribute. If attribute article_reference of type IDREF has been defined for the element type paragraph, the start-tag <paragraph article_reference="A123"> can appear only if there is an ID type attribute with the value A123 in the document. If the attribute article_reference is defined with type IDREFS, the start-tag <paragraph article_reference="A123 A567"> can appear if

there is an ID type attribute with value A123 and another ID type attribute with value A567. A common practice is to give the name id to ID type attributes.

The attribute types ID and IDREF(S) offer a restricted tool for unique identification and cross-referencing. As just described, since each value of type ID must be unique within the document, the same value cannot be used for different element types nor for different attributes. Furthermore, in a valid XML document, the value associated with any attribute of type IDREF or included in a list of values for an attribute of type IDREFS must match the value associated with an attribute of type ID *within the same document*. Thus, the mechanism can be used only inside a single document, not across a set of documents.

In the attribute definition using an *enumerated type*, the attribute values to be accepted in valid documents are specified as part of the declaration. For example, if we would like to associate the root element of the document type report with an attribute phase to show the state of progress, the values could be given in the following way:

```
<!ATTLIST    report    phase    (draft | comments_requested | final)   #REQUIRED >
```

Instead of declaring the attribute to be mandatory, the declaration

```
<!ATTLIST    report    phase    (draft | comments_requested | final)   “draft” >
```

specifies that the value draft is the default value. In this case, if the attribute is not present in a report element, then the processor supplies the value draft.

Example 2.8 Two attributes are declared for the element type clock as follows:

```
<!ATTLIST    clock     setting    CDATA       #IMPLIED
             alarm     (yes | no | dual)  "yes"   >
```

The attribute setting is a string type attribute without any default value. Attribute alarm is of enumerated type with three valid values: yes, no and dual. Because of the specified default, if there is no attribute alarm explicitly included in the start-tag of a clock element, the XML processor must attach the attribute with value yes to the element. The following hypothetical product description provides an example of the use of these attributes:

```
<product >
   <mfg>Nokia</mfg><model>8890</model>
   <description> Intended for EGSM 900 and GSM 1900 networks.</description>
   <clock setting= "nist" alarm = "no"/>
</product>
```

The constraining mechanism provided by DTDs for attribute types is very limited. Nevertheless, if an application uses the DTD mechanism as the schema language, the existence of some limited facility for attribute typing and the lack of facilities for typing character data found in element content may be influential when deciding whether to use elements or attributes for some data.

In addition to application-defined attributes, there are two predefined attribute names, xml:space and xml:lang, available for use in XML documents. The prefix xml indicates that the names are reserved by the XML specification. Nevertheless, in valid documents these predefined attributes, like any other, must be declared if they are used. Attribute xml:space signals an intention that white space should be preserved by applications in the element, and its type must be an enumerated type with values default and preserve. The attribute xml:lang is used to specify the language of the contents and of other attribute values of an element. The values of the attribute must be a subset of the codes defined in the specification IETF RFC 1766, which uses abbreviations such as en, fr, fi, en-GB, and en-US to denote a language. For example, these attributes could be declared for the type poem as follows:

```
<!ATTLIST      poem      xml:space (default | preserve)    "preserve"
               xml:lang  (fi | en)     "fi"     >
```

Before an attribute's value is passed to an application or checked for validity, the XML processor normalizes it by applying the algorithm given in Sect. 3.3.3 of the XML specification. The normalization converts character and entity references and white space to a standard form in which references are replaced by their values. Character and entity references and their replacement are considered further in Sect. 2.4.5 below.

2.4.4 Entity and Notation Declarations

In a valid document, all entities must be declared before they are used. The declaration gives a name and, in the case of internal entities, a value for the entity. For external entities a reference to the external file must be provided. The value of the internal entity given in the declaration is called a *literal entity value*.

The declarations for parameter and general entities are distinguished by the presence or absence of one character: a parameter entity is introduced by a percent sign (%) before the name of the entity, whereas a general entity is not. As examples, consider first the following general entity declaration:

```
<!ENTITY xml-spec "Bray, T., Paoli, J., Sperberg-McQueen, C.M., & Maler, E. (Editors),
Extensible Markup Language (XML) 1.0 (Second Edition),
W3C Recommendation 6 October 2000">
```

The 156-character string starting with "Bray" and ending with "2000" is given the name xml-spec. Once declared as a general entity, it is usable as often as desired within the elements of a document, but not in the DTD. On the other hand, the character % in the following declaration shows that it is declaring a parameter entity:

```
<!ENTITY % chapter_attributes
    "author     NMTOKEN          #IMPLIED
    date        CDATA            #REQUIRED"      >
```

This entity is named chapter_attributes and is defined to represent two attribute definitions that may be needed repeatedly in the DTD. Unlike general entities, parameter entities are used in DTDs only and not elsewhere in documents.

Since parameter and general entities are recognized in different contexts, and they use different forms of reference, they also occupy different namespaces. This means that a general entity and a parsed entity with the same name are two distinct entities.

The literal value of internal entities is not necessarily the value by which the name of the entity is replaced at the place where it is used. As stated above, some processing of entity and character references in the literal may be needed before the replacement; this is further discussed in Sect. 2.4.5.

Example 2.9 The following examples of parameter entity declarations appear in the XHTML specification [20].

```
<!ENTITY % URI "CDATA">
    <!-- a Uniform Resource Identifier, see [RFC2396] -->
<!ENTITY % UriList "CDATA">
    <!-- a space separated list of Uniform Resource Identifiers -->
<!ENTITY % StyleSheet  "CDATA">
    <!—stylesheet data -->
<!ENTITY % Text  "CDATA">
    <!--used for titles etc. -->
<!-- core attributes common to most elements
      id              document-wide unique id
      class           space separated list of classes
      style           associated style info
      title           advisory title/amplification
-->
<!ENTITY % coreattrs
      "id             ID                      #IMPLIED
      class           CDATA                   #IMPLIED
      style           %StyleSheet;            #IMPLIED
      title           %Text;                  #IMPLIED" >
<!ENTITY % heading  "h1 | h2 | h3 | h4 | h5 | h6">
<!ENTITY % list  "ul | ol | dl">
```

For external entities a *system identifier* is given to allow the XML processor or its client application to locate the entity. For example, an external entity section1 could be declared to be found at the location given:

```
<!ENTITY  section1  SYSTEM  "http://www.cs.jyu.fi/opetus/xml/section1.xml">
```

In this case the system literal is an absolute URI reference. It can also be a relative URI, to be converted to an absolute URI reference by the XML processor. The string given as a system literal may also contain characters intended to be escaped before a URI can be used to retrieve the referenced entity. The handling of relative URIs and escape characters is described in Sect. 4.2.2 of the XML specification. The system identifier is sometimes preceded by a public identifier, which is intended to provide a label generally understood by the applications. The XML processor may use any combination of the public identifier and system identifier, as well as some additional information, in attempting to retrieve the entity's content.

The previous example declared section1 to be an external parsed entity. The declaration of an *unparsed* entity requires the identification of the file format for the entity, which is specified by a notation name introduced by a *notation declaration*. The notation declaration provides an external identifier that allows the XML processor or its client application to locate a suitable application capable of processing data in the given format. Notation names are used not only in entity declarations to specify the format of unparsed entities, but can also appear in attribute-list declarations for enumerated attribute types. A notation for using pictures in gif format could be introduced by the following notation declaration:

```
<!NOTATION gif PUBLIC
  "-//ISBN 0-7923-9432-1::Graphic Notation//NOTATION CompuServe Graphic
  Interchange Format//EN" >
```

The notation can then be used to declare an external unparsed entity as follows:

```
<!ENTITY picture1 SYSTEM "../pictures/scenery.gif" NDATA gif>
```

The presence of the token NDATA followed by the token gif declares that picture1 uses the notation gif, and together with the notation declaration notifies the XML processor that it should invoke an appropriate graphics handler.

Five general entities, amp, lt, gt, apos, and quot, are predefined in the XML specification and therefore need not, and must not, be declared in a DTD. These are used to escape the markup delimiters (ampersand (&), left angle bracket (<), right angle bracket (>), apostrophe ('), and quotation mark ("), respectively).[3]

2.4.5 XML Processor Treatment of Entities and References

A validating processor includes an entity in the physical structure of an XML document if it is the root entity, an external subset of the document type definition, or an entity referred to by its name in an entity included in the physical structure. A non-validating processor does not necessarily read external entities.

[3] By default, curly apostrophes and quotation marks are commonly used in place of straight ones in documents prepared by word processors. However, these marks are not accepted by XML processors and are a common cause of parsing errors when examples are copied for XML parsing.

Table 2.2 Contexts for referencing entities and characters

Referencing type	Contexts
Unparsed entity reference	• As attribute value in a start-tag or in an attribute definition
Parameter entity reference	• Document type definition • Entity value (for an entity used in a document type definition)
General entity reference	• Element content • Attribute value either in a start-tag or in an attribute definition • Entity value

Unparsed entities, which are always also external entities, are referenced by giving the name as an attribute value for some element. In a valid document the referencing attribute must have been declared as an entity type attribute (ENTITY or ENTITIES). Unparsed entities are not intended for processing by an XML processor. Instead, the processor merely passes the identifiers for each entity and the associated notation to the application.

On the other hand, parsed entities are processed by the XML processor, which replaces the entity name by the entity value, as described below. References to parsed entities, called *entity references*, may appear outside attributes. Parameter entity references always begin with a percent sign (%) and terminate with a semicolon(;), and the context of a parameter entity reference is always the document type definition. General entity references always start with an ampersand (&) and end with a semicolon. For example, &xml-spec; references the general entity xml-spec, and %chapter_attributes; designates the parameter entity chapter_attributes. Recall that a document may contain both a parameter entity and a general entity named title, for example; their uses are distinguished by the syntax of the references (%title; versus &title;) and whether they appear within a DTD or within the content of an element.

Table 2.2 summarizes the contexts in which invocations of unparsed entities and entity references might appear.[4]

To understand the processing of parsed entities and their references, it is important to distinguish between two kinds of entity content: the literal entity value and the replacement text. As mentioned in the previous section, the quoted string given in the declaration of an internal entity is called a literal entity value. The literal entity value may contain character, parameter entity, and general entity references. The *replacement text* for a parsed internal entity reference is derived by replacing character references by their character values and parameter entity references by their replacement texts.

[4] Character references are syntactically similar to general entity references and can appear in the same contexts. However, character references are not parsed as described in this section. See instead Sect. 2.3.3.

Example 2.10 Consider again some of the entity declarations from Example 2.9:

```
<!ENTITY % StyleSheet  "CDATA">
    <!—stylesheet data -->
<!ENTITY % Text  "CDATA">
    <!-- used for titles etc. -->
<!ENTITY % coreattrs
     "id            ID              #IMPLIED
      class         CDATA           #IMPLIED
      style         %StyleSheet;    #IMPLIED
      title         %Text;          #IMPLIED"  >
```

The literal entity value of the entity StyleSheet and of the entity Text is CDATA, which is also the replacement text for those entities. The literal entity value of the entity coreattrs is

```
id      ID                  #IMPLIED
class   CDATA               #IMPLIED
style   %StyleSheet;        #IMPLIED
title   %Text;              #IMPLIED
```

whereas the replacement text is derived by replacing the parameter entity references by their replacement texts:

```
id      ID              #IMPLIED
class   CDATA           #IMPLIED
style   CDATA           #IMPLIED
title   CDATA           #IMPLIED
```

Any nested general entity references are left unexpanded when deriving such replacement text; instead expansion of general entity references, replacing each reference by the value of the entity referenced, takes place only when the entity reference is encountered in element contents and in attribute values.

For a parsed *external* entity, the literal entity value is the exact text contained in the entity (external file) and the replacement text is the content of the entity after stripping the text declaration, if there is one, but *without* any replacement of character or parameter entity references.

The *replacement* text for a parsed entity is regarded as an integral part of the document in place of its entity reference. The detailed way an XML processor treats an entity and its references depends on the type of references, their contexts, and the type of the processor. Non-validating processors need not deal with external entities, nor are they obligated to read and process entity declarations occurring within parameter entities. Hence a non-validating processor is not necessarily aware of all entity declarations. The details of the treatment of entities and references are described in Sect. 4.4 of the XML specification.

Example 2.11 The XHTML specification also contains the following entity declarations:

```
<!ENTITY % special.pre        "br | span | bdo | map ">
<!ENTITY %special             "%special.pre; | object | img ">
<!ENTITY % fontstyle          "tt | i | b | big | small ">
<!ENTITY % phrase
   "em | strong | dfn | code | q | samp | kbd | var | cite | abbr | acronym | sub | sup ">
<!ENTITY % inline.forms       "input | select | textarea | label | button">
<!ENTITY % misc.inline        "ins | del | script">
<!ENTITY % inline
   "a | %special; | %fontstyle; | %phrase; | %inline.forms;">
<!ENTITY % Inline"(#PCDATA | %inline; | %misc.inline;)*">
```

The literal value of the entity Inline is

```
(#PCDATA | %inline; | %misc.inline;)*
```

The replacement text is derived by replacing the references to parameter entities inline and misc.inline by their replacement texts. The literal value for the entity inline

```
a | %special; | %fontstyle; | %phrase; | %inline.forms;
```

must therefore be transformed to its replacement text, which requires examining its four referenced entities. When processing the literal value of the entity special

```
%special.pre; | object | img
```

the processor encounters a reference to the entity special.pre whose replacement text is

```
br | span | bdo | map
```

After expanding everything, the replacement text of the entity inline is found to be:

```
a | br | span | bdo | map | object | img | tt | i | b | big | small | em | strong | dfn | code | q |
samp | kbd | var | cite | abbr | acronym | sub | sup | input | select | textarea | label | button
```

After also expanding the entity %misc.inline;, the replacement text of the entity Inline is found to be:

```
(#PCDATA | a | br | span | bdo | map | object | img | tt | i | b | big | small | em |
strong | dfn | code | q | samp | kbd | var | cite | abbr | acronym | sub | sup | input |
select | textarea | label | button | ins | del | script)*
```

2.4.6 XML Schema

The document type definition mechanism provided by DTDs is just one means to constrain XML data. Several other definition languages, called schema languages, have also been defined for XML. Software supporting each such language is able to check the validity of an XML document against the definitions of acceptable data types and structures declared by the document designers.

A language widely adopted for many applications is *XML Schema*, also known as *XSD*, developed by W3C and described in three parts: a primer summarizing the language and providing several examples [14], the specification for describing compound structures [21], and the specification for the 44 pre-defined atomic data types available in XSD [2]. This provides richer constraining mechanisms than those available through a DTD, and they can all be applied to both attribute values and element contents.

2.4.6.1 Overview

We begin by comparing some core features of XSD to those of DTDs, divided into the following areas:

- XML model
- Types
- Syntax
- Namespaces

Following that, we demonstrate how elements are declared and types defined, and then how attributes are declared.

XML model. As described in Sect. 2.3, the core concept of XML is a document with both physical structure and logical structure. The markup in an XML document provides information for both of these structures, and DTDs include definition capabilities for them both. A DTD is used by validating XML processors to assess the validity of XML documents with respect to element type declarations, attribute list declarations, entity declarations, and notation declarations.

As will be explained in Sect. 2.5.2 below, the nesting of elements in XML imposes a computational structure known as a *tree*. Based on this relationship, the XML Information Set (Infoset) model describes an XML document as an abstract tree structure consisting of 11 kinds of nodes called information items [12]. Unlike a DTD, an XSD instance is not intended to assess the validity of marked up XML documents but rather the validity of element and attribute information items as they are defined by the Infoset model. Thus even though both XSD and DTDs are languages that constrain the contents of XML documents, the target for XSD is different from that for DTDs.

Types. The most significant improvement of XSD over DTDs is the introduction of a rich typing system, allowing designers to declare restricted domains of values for each of the elements and attributes in their documents.

Consider first the following DTD declarations for a multilingual street name catalogue:

```
<!ELEMENT   streetCatalogue              (street+)>
<!ELEMENT   street                       (streetName+)>
<!ELEMENT   streetName                   (#PCDATA)>
<!ATTLIST   streetName     lang    NMTOKEN   #REQUIRED>
```

This example includes declarations for three *element types* and one declaration associating an *attribute name* with an *attribute type*. For each street there may be several street names, and the language of each street name is provided using the lang attribute.

In a DTD, although the content of elements having children can be structurally constrained, elements without any child elements always contain arbitrary strings of character data (declared as #PCDATA). In the above example, the length of a streetName cannot be constrained. In other situations where ages, monetary amounts, or dates are stored as element values, no constraints can be declared to ensure that valid integers or dates are actually stored. Attribute values can be constrained by the attribute type, but the choice of types is very limited, as described in Sect. 2.4.3 above.

In XSD the term *type* always refers to the set of values allowable for an element or attribute, never to its name. The typing system in XSD allows document designers to specify elements and attributes that include all the constraints available in DTDs, but many additional forms of constraint are also available. Types can be *simple* (such as integer, string, date, and time), which will be described in more detail in Sect. 5.1, or they can be structures with arbitrarily many sub-components (so-called *complex types*).

Elements and attributes are constrained by *schema components* in the form of definitions and declarations. *Definition* components specify *types* as sets of possible values, including both atomic and structured values. On the other hand, element, attribute, and notation *declarations* are used to enable elements, attributes, and notations with the specific names to appear in document instances and to constrain the contents of each appearance to conform to a defined type.

The declaration for an element or attribute may include a type definition directly or it may refer to the type by the type name. In the former case the type is *anonymous*. In the latter case the definition of the type with the given name may be included in the same schema, or the name can refer to a type defined in another schema or in the XSD specification itself. Note that the content specifications of element type declarations of DTDs correspond to XSD's anonymous type definitions.

Syntax. Whereas the syntax for DTDs is especially defined for markup declarations, XSD uses XML's element and attribute notation. For example, elements named element and attribute are used to declare elements and attributes, respectively, and elements named complexType and simpleType are used to define complex types and simple types (those without attributes and subelements), respectively. (Examples will be given in Sect. 2.4.6.2 below).

Schema instance files are usually named with the .xsd file extension when they are intended to be processed by XSD-aware software. However, XSD schema instances are also special cases of XML documents, and therefore they can be processed by general XML software. The extension .xml is used instead when the file is to be processed by general purpose XML software.

Namespaces. The element and attribute names in a DTD must appear in the same form as in the document validated against the DTD. Prefixed names denoting namespaces can be used in a DTD, but namespaces cannot be declared there since they can be declared only in the start-tags of elements. Therefore two documents that use identical namespaces but with different prefixes cannot be validated with a single DTD.

Parameter entities provide an alternative method to support reuse of declarations in DTDs. Reuse is also supported by modularization methods enabling the creation of schemas from well-defined sets of elements and attributes. Examples of these methods are presented by Eve Maler and Jeanne El Andaloussi in their extensive book on developing SGML DTDs [19] and in the descriptions of XHTML [1] and the Text Encoding Initiative (www.tei-c.org). These alternative methods, however, do not solve the problem of name collisions.

XSD supports the use of namespaces in several ways. The XSD syntax, being a subset of XML, allows the declaration of namespaces in the start-tag of any element. The vocabulary declared and defined in a schema forms a *target namespace*, which therefore includes the names declared for all elements and attributes, as well as of the names defined for types in the schema. For example, the following lines might start a schema:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:school="http://www.example/schoolNames"
        targetNamespace="http://www.example/schoolNames">
        <element name="beginDate" type="date" />
```

The first line declares the XSD namespace as the default namespace for the schema itself. This allows the use of the names from the namespace http://www.w3.org/2001/XMLSchema without any prefix. The second line associates the prefix school with the namespace name http://www.example/schoolNames. The third line declares this particular namespace as the target namespace of the schema. As a result, that namespace is populated by the names declared for elements and attributes in the schema, as well as by the names defined for types in the schema. The last line declares the beginDate element to have the built-in type date. The name beginDate belongs to the target namespace, but the element names schema and element, the attribute names name and type, and the type name date are all taken from the namespace http://www.w3.org/2001/XMLSchema.

It is worth noticing that the namespace concept of XSD extends the XML Names specification [4] by including type names as well as names of elements and attributes. Thus the example above shows that the namespace http://www.w3.org/2001/XMLSchema includes the element names element and attribute and the attributes name

and type, as well as the names of the built-in types such as date. Notice also that the names of types are not used as element or attribute names in schemas but instead as attribute values.

A schema defining a particular target namespace may be divided into several schema documents by using the include element to specify the location of the file from which schema components are included. In such a case, there is a single declared target namespace for all of the schema documents, and that target namespace is populated by the names declared and defined in all the documents.

The reuse of types is facilitated by the import element, which identifies the target namespace for the imported types. For example, the types defined in the target schema http://www.example/schoolNames could be reused by another schema by including the following schema element:

```
<import namespace="http://www.example/schoolNames"/>
```

2.4.6.2 Declaring Elements and Defining Types

Consider defining a schema for the following simple XML document:

```
<student>
    <name>Steve  Chung</name>
    <age>23</age>
    <phone>416-982-1111</phone>
</student>
```

The student element contains three subelements: name, age, and phone, each of which consists of character data. A possible DTD for the data might include four element type declarations as follows:

```
<!ELEMENT    student    (name, age, phone)>
<!ELEMENT    name       (#PCDATA)>
<!ELEMENT    age        (#PCDATA)>
<!ELEMENT    phone      (#PCDATA)>
```

An XSD definition for the same data is shown in Fig. 2.3, where the schema components are contained in the schema element. As in the example in the previous section, the XSD namespace http://www.w3.org/2001/XMLSchema is first declared as the default namespace, and then the prefix school is associated with the namespace http://www.example/schoolNames, which is also declared as the target namespace. The schema includes four element declarations, each specified in an element named element that associates a name, given with the attribute name, with a type.

Since name, age, and phone do not have any attributes or child elements, they can be declared to have a simple type. For example, the element

```
<element name="phone" type="string" />
```

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:school="http://www.example/schoolNames"
        targetNamespace="http://www.example/schoolNames">
    <element name="student">
        <complexType>
                <sequence>
                    <element name="name" type="string" />
                    <element name=”age” type=”positiveInteger”/>
                    <element name="phone" type="string" />
                </sequence>
        </complexType>
    </element>
</schema>
```

Fig. 2.3 A schema with an anonymous complex type definition

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:school="http://www.example/schoolNames"
        targetNamespace="http://www.example/schoolNames">
    <complexType name="PersonalData">
        <sequence>
            <element name="name" type="string" />
            <element name="age" type="positiveInteger"/>
            <element name="phone" type="string" />
        </sequence>
    </complexType>
    <element name="student" type="school:PersonalData"/>
</schema>
```

Fig. 2.4 A schema with a named complex type definition

declares the element name phone to have the built-in type string. Note, however, that unlike DTDs, XSD allows the age to be constrained to contain a string that represents a positive integer.

Unlike the other three elements, the student element has subelements, and it must therefore be declared as a complex type. In Fig. 2.3 the type is declared using an *anonymous type definition*: the elements complexType and sequence define a sequence structure consisting of the elements name, age, and phone.

Instead of using an anonymous type definition, an alternative is shown in Fig. 2.4, where a complex type named PersonalData has been defined separately from the student element declaration. The element declaration refers to this type by its name, prefixed by the name of the target namespace.

XSD allows document designers to declare new types that are derived from other atomic or complex types. For example, the type of the element phone in these examples was constrained to character strings, but any strings are allowed as element

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:school="http://www.example/schoolNames"
        targetNamespace="http://www.example/schoolNames">
    <element name="name" type="string" />
    <element name="phone" type="string" />
    <complexType name="PersonalData">
        <sequence>
            <element ref="school:name"/>
            <element name=”age” type=”positiveInteger”/>
            <element ref="school:phone" minOccurs="0" />
        </sequence>
    </complexType>
    <element name="student" type="school:PersonalData"/>
</schema>
```

Fig. 2.5 A schema with references to global elements

content. Instead, a designer can constrain the content to conform to a particular pattern by declaring a *restriction* of the type string as follows:

```
<element name="phone">
  <simpleType>
    <restriction base="string">
      <pattern value="\d{3}-\d{3}-\d{4}"/>
    </restriction>
  </simpleType>
</element>
```

The restriction is specified with a *facet* called a *pattern* from the *base type* string. The pattern "\d{3}-\d{3}-\d{4}" describes a string where three digits are followed by a hyphen, three digits, a hyphen, and four digits. With such a declaration for the element phone, an XSD-aware validator will reject phone elements having content that does not match the pattern.

Complex types can also be constrained. For example, a new type can be defined as a restriction of PersonalData specifying that the age be between 16 and 25. Alternatively, a new complex type can be defined as an *extension* of PersonalData that also includes an e-mail address.

Document designers may wish to use an element declaration in several places with in a schema or in several schemas. To refer to a declaration, it must be declared as global by placing it as a child of the schema element. For example, the student element is a *global* declaration in Figs. 2.3 and 2.4. On the other hand, the name, age, and phone elements in those schemas are *local* declarations because they appear inside complex type definitions.

A global element can be referenced from other declarations using the ref attribute. For example, the element declarations in the complex type definition in Fig. 2.5 use the attribute ref to refer to the global name and phone declarations, whereas age remains local. Notice that the type of the elements are not repeated when referring

to a global declaration. In contrast to XSD, all elements in DTDs are global (their names can be used as references in other element type declarations); there is no corresponding mechanism to declare local elements.

In Fig. 2.5 the element referencing school:phone also includes a second attribute with the name minOccurs. With the value 0, this attribute constrains the number of occurrences of phone elements in the student element to be greater than equal to 0. Similarly, XSD provides the attribute maxOccurs to declare the maximum number of repetitions of an element. The value of the attributes minOccurs and maxOccurs may be any positive integer, or the term unbounded indicating that there is no maximum number of occurrences. The omission of either these attributes corresponds to including the attribute with a value of 1. Thus, in Fig. 2.5 school:name and age must occur exactly once (at least once and at most once), whereas school:phone may occur either 0 times or one time. In a DTD these constraints are expressed as

```
<!ELEMENT    student          (name, age, phone?)>
```

The attributes minOccurs and maxOccurs enable more precise constraints on the number of occurrences than allowed by a DTD's content model using the symbols ?, +, and *. For example, to allow at most five phone numbers, the element declaration would be written as follows:

```
<element ref="school:phone"  minOccurs="0"  maxOccurs= ="5"  />
```

2.4.6.3 Declaring Attributes

Instead of encoding all data as element content, let us examine how attributes are declared in XSDs. Continuing with the earlier example, assume that the area code is given as attribute value, instead of including it as part of the whole phone number:

```
<student>
      <name> Steve  Chung</name>
      <age>23</age>
      <phone area_code="416" >982-1111</phone>
</student>
```

In a DTD this change would require adding an attribute list declaration:

```
<!ELEMENT        student        (name, phone)>
<!ELEMENT        name           (#PCDATA)>
<!ELEMENT        age            (#PCDATA)>
<!ELEMENT        phone          (#PCDATA)>
<!ATTLIST        phone     area_code  CDATA    #IMPLIED>
```

Note that in contrast to element declarations, all attributes in a DTD are declared local by associating them with an element name. In XSD it is possible to declare attributes to be global as well as local.

Using XSD, the phone element must have a complex type to enable it to have an attribute. Figure 2.6 shows how an appropriate complex type can be derived from a

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:simpleType name="phoneNumberType">
      <xsd:restriction base="xsd:string">
         <xsd:pattern value="\d{3}-\d{4}"/>
      </xsd:restriction>
   </xsd:simpleType>

   <xsd:simpleType name="areaCodeType">
      <xsd:restriction base="xsd:string">
         <xsd:pattern value="\d{3}"/>
      </xsd:restriction>
   </xsd:simpleType>

   <xsd:element name="student">
      <xsd:complexType>
         <xsd:sequence>

            <xsd:element name="name" type="xsd:string" />
            <xsd:element name="age" type="positiveInteger"/>
            <xsd:element name="phone">
               <xsd:complexType>
                  <xsd:simpleContent>
                     <xsd:extension base="phoneNumberType">
                        <xsd:attribute name="area_code" type="areaCodeType"/>
                     </xsd:extension>
                  </xsd:simpleContent>
               </xsd:complexType>
            </xsd:element>

         </xsd:sequence>
      </xsd:complexType>
   </xsd:element>
</xsd:schema>
```

Fig. 2.6 A schema with an attribute declaration

simple type. (For variety, the schema has also been altered such that no target namespace has been declared – i.e., the default namespace is used for the target – and the namespace http://www.w3.org/2001/XMLSchema is explicitly associated with the prefix xsd).

In this example, two simple types have been defined to constrain the phone number and the area code respectively: the phoneNumberType restricts a string to the form where three digits are followed by a hyphen and four digits, and the areaCode-Type restricts a string to be three digits. The type of phone is an anonymous complex type. The element simpleContent indicates that the content of the element contains only character data, no sub-elements. The complex type is derived as an extension from the base type areaCodeType by adding an attribute with name area_code and type areaCodeType.

2.4.6.4 Extended XSD Example

Before concluding this section, we return to the earlier example of a rhyme to illustrate grouping, optionality, alternatives, and iteration in XML Schema. For simplicity, the example of alternative content shown here is via an enumerated type (for xml:lang); more general alternatives are defined using the element <xs:choice> in place of <xs:sequence>.

Example 2.12 XSD declarations for rhymes that parallel the DTD declarations in Example 2.4:

```
<xs:schema elementFormDefault="qualified" attributeFormDefault="unqualified"
                xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="line" type="xs:string" />
<xs:element name="rhyme">
    <xs:complexType mixed="false">
        <xs:sequence minOccurs="1" maxOccurs="1">
            <xs:complexType mixed="false">
                <xs:element ref="line" minOccurs="1"  maxOccurs="unbounded"/>
            </xs:complexType>
        </xs:sequence>
        <xs:attribute name="xml:lang" use="optional">
            <xs:simpleType>
                <xs:restriction base="xs:language">
                    <xs:enumeration value="fi"/>
                    <xs:enumeration value="en"/>
                </xs:restriction>
            </xs:simpleType>
        </xs:attribute>
        <xs:attribute name="author" type="xs:string" use="optional"/>
    </xs:complexType>
</xs:element>
<xs:element name="rhymecollection">
    <xs:complexType mixed="false">
        <xs:sequence minOccurs="1" maxOccurs="1">
            <xs:element name="title" type="xs:string" minOccurs="0" maxOccurs="1"/>
            <xs:element ref="rhyme" minOccurs="1"  maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
 </xs:element>
</xs:schema>
```

In Example 2.12, a rhyme is specified as being a non-empty, unbounded sequence of lines with no interleaved text (mixed="false"). Each line is itself a simple string. A rhyme may optionally have attributes as follows: xml:lang is an enumerated type that restricts the built-in type xs:language, and author can take any string as its value.

A rhymecollection starts with an optional title and then has one or more elements of type rhyme, declared earlier.

In summary, XSD is more expressive than DTDs: it can be used to specify constraints that cannot be described using a DTD. The availability of many atomic types allows element content to be constrained rather than merely being declared generically as #PCDATA. In addition, minOccurs and maxOccurs can take any numeric values, not merely 0, 1, or unbounded. The items in a sequence are constrained to be in a fixed order (as in DTDs, where, for example, name, age, and phone number may all be required in that order), but by using <xs:all> instead of <xs:sequence> the component elements are allowed to appear in arbitrary order (name, age, and phone number must all be provided but any of the six possible orderings are acceptable). Elements of any type can be declared with the attribute nillable="true" to allow any instance of that element to have empty content when xsi:nil="true" is included among its attributes (assuming that somewhere in its enclosing context xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" has been declared); thus the fact that a value is missing, unknown or not applicable can be recorded for elements that must be present and otherwise could not be empty (such as integers, dates, or complex types with required sub-elements). Finally, any element can be declared to have a value that is unique with respect to all other elements in the same context; for example, employee id numbers can be declared to be non-repeating throughout the document, non-repeating within each organization listed in the document, or non-repeating within each division in each organization.

Finally, DTDs provide the attribute types ID, IDREF, and IDREFs to support links between elements within a document. XSD extends this to allow the definition of keys (similar to relational database keys, but with respect to a particular context within each document) and references (of type keyref) to those keys.

2.4.7 RELAX NG

A third data definition language is *RELAX NG*, developed by James Clark and Murata Makoto through OASIS and an ISO standard (ISO/IEC 19757–2) since December 2003 [10, 11]. RELAX NG is simpler than XML Schema, but it too includes a richer collection of data types than available in DTDs as well as support for namespaces.

As will be explained in Sect. 2.5.2 below, the nesting of elements in XML imposes a structure known in computer science as a *tree*. RELAX NG is designed to constrain the trees that are represented by XML documents rather than capabilities for constraining the document text directly. In RELAX NG, therefore, the content models for elements, as well as for the sets of valid values for attributes, are modeled as *tree-regular grammars*, a formalism similar to EBNF (as introduced in

Sect. 2.1) but describing trees rather than strings. RELAX NG is not able to describe all the constraints describable with XSD, but it is more expressive than XSD in specifying unordered and mixed content.

Grammars in RELAX NG can be expressed using XML structures (quite similar in style to that used in XSD), but a more compact syntax is also available, as illustrated in the following example:

Example 2.13 RELAX NG declarations for rhymes that parallel the DTD declarations in Example 2.4 and the XSD declarations in Example 2.12:

```
grammar {
    start = RhymeCollection
    RhymeCollection = element rhymeCollection { element title { text} ?, Rhyme+ }
    Rhyme = element rhyme {
                    attribute xml:lang { ("fi" | "en") },
                    attribute author {text},
                    Line+
      }
    Line = element line { text }
    }
```

2.5 Processing Models

As explained in Sect. 2.2, software that needs to read or modify data stored in an XML document accesses that data via an XML processor. The responsibility of the processor is to distinguish markup from content, to ensure that the document is well-formed, possibly to ensure that the document satisfies various validity constraints, to use the markup to identify individual units of content as well as the relationships between those units, and to identify suitable applications to handle non-textual components. This information is made available to the application software according to a pre-determined protocol that dictates the form of communication between the XML processor and the XML application. Such a protocol is typically embedded into an *application program interface*, or *API*, which is a set of functions that support the communication between the participating pieces of software.

There are two major protocols that are used by XML processors. In the first one, the XML document is viewed as a string of beads: a linear structure formed by interleaved markup and content. In the alternative protocol, the XML document is viewed as a bunch of grapes: a hierarchical structure matching the nested nature of the markup, with units of content situated at various points at the lowest levels of the hierarchy [22]. Managing text via these two models is discussed in the remainder of this section.

2.5.1 Stream Processing

Consider the XML element

```
<date><month>December</month><year>1654</year></date>
```

The simplest interpretation of the structure and content is to view this as a stream of tokens, where each token carries either a unit of markup or a unit of content:

Opening tag:	date
Opening tag:	month
String:	December
Closing tag:	month
Opening tag:	year
String:	1654
Closing tag:	year
Closing tag:	date

Tokens from this stream can be passed from the XML processor to the XML application in the order in which they appear, and it is up to the application software to handle the information conveyed by the tokens and their ordering in an appropriate manner to achieve its goals.

Applications that adopt this form of processing are typically based on SAX, the "Simple API for XML" [7]. With SAX, the XML processor signals the occurrence of each token in turn by calling an appropriate function, depending on the type of token. A suitable token handler must be written by the XML application programmer as the body of each designated function. For example, when a SAX parser encounters an opening tag, it calls the function startElement, passing parameters that contain the tag name, information to resolve the namespace, and the list of attribute-value pairs. Assuming that the software is written in the Java programming language, the XML application programmer must implement the function

```
public void startElement (String uri, String name, String qName, Attributes atts){ … }
```

to handle a start-tag whenever it is encountered in the XML document. The following table lists the set of token types that may occur in a SAX stream:

Tokens from a DTD	
notationDecl	unparsedEntityDecl
Tokens from the body	
startDocument	endDocument
startElement	endElement
startPrefixMapping	endPrefixMapping
characters	ignorableWhitespace
processingInstruction	skippedEntity

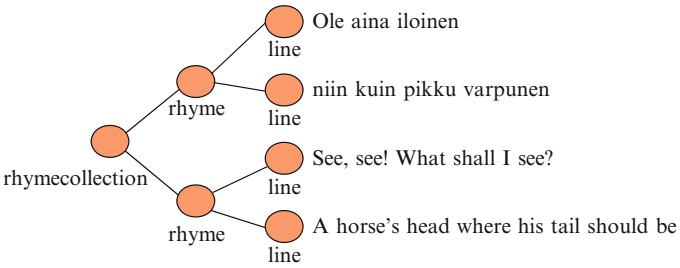

Fig. 2.7 XML document as a tree

2.5.2 Tree Processing

It is often important to consider the context of a fragment of an XML document in order to process it appropriately. In such circumstances, it is convenient to have easy access to the nesting structure implied by the markup. Thus, instead of viewing an XML document as a stream of tokens, some application programs are better served by viewing it as a hierarchical structure, and more specifically as an ordered, rooted tree. The document element is interpreted as the root of the tree, and each non-root element is interpreted as a child of the node corresponding to the element in which it is contained. An XML processor supporting the tree-based protocol for communicating the information about an XML document to an XML application must provide information about the order of sibling elements to that application.

Consider again the following XML document with two rhymes, one written in Finnish, another in English.

```
<rhymecollection>
    <rhyme>
    <line>Ole aina iloinen</line>
    <line>niin kuin pikku varpunen</line>
    </rhyme>
    <rhyme>
    <line>See, see! What shall I see?</line>
    <line>A horse's head where his tail should be</line>
    </rhyme>
</rhymecollection>
```

The corresponding tree structure can be depicted as shown in Fig. 2.7. In the tree the children of a node are ordered from top to bottom and reflect exactly the same order as they appear (left to right) in the corresponding elements in the tagged text.

Figure 2.7 shows only the element structure. Tree models for XML documents have been developed in four different specifications proposed through W3C: the XML Information Set (Infoset) model [12], the XML Path Language (XPath 1.0) data model

[9], the Document Object Model (DOM) [18], and the XQuery 1.0 and XPath 2.0 Data Model [15]. All four cover more than just the structure of elements. Each model defines a set of node types, including types for the root of a document as well as for elements, attributes, comments, and processing instructions. However there are some subtle differences between the models. For example, if a tree is created on the basis of the Infoset model, there is a distinct node for each character in the content of an element (thus resulting in 16 child nodes for the first line node in the above example), whereas the XQuery/XPath data model permits multiple-character strings in a content node and specifies that two adjacent sibling nodes may not both contain text (thus resulting in only one child node for each line node in the above example). Furthermore, although attributes are represented by nodes in all four models, there are some subtle differences in the way in which the four models account for the fact that there is no order defined among attributes of an element in the XML specification. The existence of competing tree models for XML data has caused some inconsistencies and incompatibilities in XML development.

The functionality available through XPath 1.0 provides an example of an application's access to an XML document using a tree-based protocol. An XPath expression is a sequence of steps in which each step is interpreted with respect to a starting node (the context node) and returns either a (possibly empty) set of nodes, a Boolean value (true or false), a number, or a character string. For example, starting from the root of the above tree, the expression rhyme/line yields a node set including all four line elements, rhyme[2]/line yields the last two line elements (i.e., the two English lines), and rhyme/line[contains(.,"niin")]/text() yields the contents of the second Finnish line, namely the string "niin kuin pikku varpunen".

More precisely, the steps are separated by slashes and each step includes an axis, a node test, and a predicate, defined as follows:

- Starting from a context node, an axis identifies a subset of nodes in the tree and imposes a linear ordering on that subset. For example, the child axis selects all element children of a node and orders them from first to last; the ancestor axis selects the parent of the node first, followed by the parent's parent, that node's parent, and so on until the root node is reached. XPath 1.0 defines 13 axes (ancestor, ancestor-or-self, attribute, child, descendant, descendant-or-self, following, following-sibling, namespace, parent, preceding, preceding-sibling, self), any of which can be specified in any step of an expression. For simplicity, an abbreviated form may be used wherein the child axis need not be specified, descendant-or-self is represented by omitting the step (thus effectively doubling the slash), parent is represented by a caret (^), and self is represented by a dot (.).
- Starting with the sequence of nodes designated by an axis, the node-test eliminates any node that does not match the specified element name or element type. For example, child::line selects only child nodes corresponding to elements named line and descendant::comment() yields all comment nodes within the subtree of the context node. When used as a node-test, an asterisk (*) matches all nodes designated by the axis.
- Starting from the sequence of nodes designated by an axis and passing the node-test, a predicate serves as an additional filter to select the subset of the nodes for

which it evaluates to *true*. As an abbreviation, if the predicate is a number, it selects the node from that position in the sequence, and if it is a path expression, it selects each node *N* for which that path expression evaluates to a non-empty set of nodes when *N* is used as the context node.

Finally, a path consisting of a sequence of steps is evaluated by determining the set of nodes from the first step, using each in turn as a context node to select nodes using the second step and forming the union of all returned node sets, and so forth. For example, starting from some context node, the XPath expression

```
chapter/section[2]//figure/^subsection
```

selects all child element nodes with name chapter, then selects all second sections that are children of those chapters, then all figures anywhere within those sections, and finally all subsections that immediately contain those figures.

2.5.3 Comparing Stream and Tree Processing

Processing an XML document as a stream of tokens allows an application to consider all components in the order that they appear in the document. Parsing a document for stream processing is straightforward and stream parsers are very efficient regardless of the size of a document. Stream processors are typically used for XML documents that are shipped from one application to another.

Tree processing requires the parser to build and maintain a nested representation of the document, usually within a computer's main memory. As a result, an application can navigate forward and backwards within the structure, climbing one branch of the tree and descending others that may have occurred before or after it in document order. The parser and the application interface are therefore somewhat more complicated, and typically document processing is restricted to documents, and their parses, that fit into main memory. Tree processors are therefore typically used for applications that require random access to sub-parts of a document, and they are more amenable to supporting XML validation, especially with respect to resolving attributes of type IDREF(S).

In spite of recognizing their advantages and disadvantages, however, it is important to remember that either type of processor is capable of preparing an XML document for use in any application.

References

1. Austin, D., Peruvemba, S., McCarron, S., Birbeck, M. (eds): XHTML™ Modularization 1.1 – Second Edition. W3C Recommendation (29 July 2010) http://www.w3.org/TR/xhtml-modularization/, Cited 10 March 2011.
2. Biron, P., Malhotra, A. (eds): XML Schema Part 2: Datatypes (Second Edition). W3C Re-commendation (28 Oct 2004) http://www.w3.org/TR/xmlschema-2/, Cited 10 March 2011.

3. Bray, T.: The Annotated XML Specification. http://www.xml.com/axml/testaxml.htm, Cited 10 March 2011.
4. Bray, T., Hollander, D., Layman, A., Tobin, R., Thompson, H.S. (eds): Namespaces in XML 1.0 (Third Edition). W3C Recommendation (8 December 2009) http://www.w3.org/TR/xml-names/, Cited 10 March 2011.
5. Bray, T., Paoli, J., Sperberg-McQueen, C.M. (eds): Extensible Markup Language (XML) 1.0. W3C Recommendation (10 February 1998) http://www.w3.org/TR/1998/REC-xml-19980210, Cited 10 March 2011.
6. Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., Yergeau, F., Cowen, J. (eds): Ex-tensible Markup Language (XML) 1.1. W3C Recommendation (4 February 2004, edited in place 15 April 2004) http://www.w3.org/TR/2004/REC-xml11-20040204/, Cited 10 March 2011.
7. Brownell, D. (ed): SAX. http://www.saxproject.org/, Cited 10 March 2011.
8. Clark, J., Pieters, S., Thompson, H.S. (eds): Associating Stylesheets with XML documents 1.0 (Second Edition). W3C Recommendation (28 October 2010) http://www.w3.org/TR/xml-stylesheet, Cited 10 March 2011.
9. Clark, J., DeRose, S. (eds): XML Path Language (XPath) Version 1.0. W3C Recommendation (16 November 1999) http://www.w3.org/TR/xpath, Cited 10 March 2011.
10. Clark, J, Murata, M.: RELAX NG Specification, Committee Specification. OASIS (3 De-cember 2001) http://www.oasis-open.org/committees/relax-ng/spec-20011203.html, Cited 10 March 2011.
11. Clark, J, Murata, M.: RELAX NG Tutorial, OASIS Committee Specification (3 December 2001), http://www.relaxng.org/tutorial-20011203.html, Cited 10 March 2011.
12. Cowan, J., Tobin, R. (eds): XML Information Set (Second Edition). W3C Recommendation (4 February 2004) http://www.w3.org/TR/xml-infoset/, Cited 10 March 2011.
13. Duerst, M., Suignard, M.: Internationalized Resource Identifiers (IRIs). The Internet Society (January 2005) http://www.rfc-editor.org/rfc/rfc3987.txt, Cited 10 March 2011.
14. Fallside, D.C., Walmsley, P. (eds): XML Schema Part 0: Primer Second Edition. W3C Recommendation (28 October 2004) http://www.w3.org/TR/xmlschema-0/. Cited 10 March 2011.
15. Fernández, M., et al. (eds): XQuery 1.0 and XPath 2.0 Data Model (XDM). W3C Recommendation (23 January 2007) http://www.w3.org/TR/2007/REC-xpath-datamodel-20070123/, Cited 10 March 2011.
16. Goldfarb, C.F.: The SGML Handbook, edited by Y. Rubinsky. Oxford University Press, Oxford, UK (1990).
17. ISO/IEC JTC1/SC34 Web Server, Information Technology – Document Description and Processing Languages. International Organization for Standardization and the International Electrotechnical Commission. http://www.ornl.gov/sgml/, Cited 10 March 2011.
18. Le Hégaret, P., et al. (eds): Document Object Model (DOM). http://www.w3.org/DOM/, Cited 10 March 2011.
19. Maler, E., El Andaloussi, J.: Developing SGML DTDs. From Text to Model to Markup. Prentice Hall PTR, Upper Saddle River, NJ (1995). Available online at http://www.xmlgrrl.com/publications/DSDTD/, Cited 10 March 2011.
20. Pemberton, S., et al.: XHTML™ 1.0 The Extensible HyperText Markup Language (Second Edition): A Reformulation of HTML 4 in XML 1.0. W3C Recommendation (26 January 2000. revised 1 August 2002) http://www.w3.org/TR/xhtml1/, Cited 10 March 2011.
21. Thompson, H.S., Bech, D., Maloney, M., Mendelsohn, N. (eds): XML Schema Part 1: Structures Second Edition. W3C Recommendation (28 October 2004) http://www.w3.org/TR/xmlschema-1/. Cited 10 March 2011.
22. Tompa, F.W.: What is (tagged) text? In: Dictionaries in the Electronic Age, Proceedings of the Fifth Annual Conference of UW Centre for the New Oxford English Dictionary and Text Research, pp. 81–93. Waterloo, Ont.: University of Waterloo (1989). Available online at http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.40.5411&rep=rep1&type=pdf, Cited 10 March 2011.
23. W3C, All Standards and Drafts. http://www.w3.org/TR/, Cited 10 March 2011.

http://www.springer.com/978-1-4614-0991-5

Communicating with XML
Salminen, A.; Tompa, F.
2011, XIV, 226 p., Hardcover
ISBN: 978-1-4614-0991-5