2 *Xml essential grammar*

In Chapter 1, we explained the motivation behind *xml* data and statement formatting, outlined the basic structure of an *xml* document, and discussed possible applications. In this chapter, we summarize the basic *xml* grammar and illustrate the implementation of a document type definition (*dtd*) or *xml* schema definition (*xsd*) for the purpose of document validation. We recall that validation is necessary only when an *xml* document structure is required to conform with agreed conventions.

We have already emphasized that computer programming experience is not necessary for generating and editing an *xml* document containing data. Consequently, this chapter can be read and understood in its entirety by a person who has never written a computer code.

2.1 Xml tags

Xml grammar and syntax are sensible and intuitive. *Xml* tags are enclosed by pointy brackets, also called angle brackets,

```
< ··· >
```

where the three dots represent an uninterrupted string of appropriate characters defining the tag name, followed by optional attributes, as discussed in Section 2.2.1. For a document to be well-formed, an opening tag, such as

```
<dianxin>
```

must be succeeded by a corresponding closing tag indicated by a slash (/)

```
</dianxin>
```

where *dianxin* is an arbitrary tag name. Words, sentences, numbers, solitary tags, or nested tags can be enclosed between an opening tag and its closure. In fact, a whole book can be written inside a single tag and its closure.

Tag names

Xml tag notation is lower and upper-case sensitive. This means that a tag named *skordalia* is not the same as the tag *Skordalia*. Tag names must be uninterrupted, that is, they may not contain empty space. When an empty space is desirable, the underscore (_) should be employed as a compromise. Tag names may not begin with any of the following strings:

```
xml  Xml xML xmL XML XmL xML XML
```

In addition, tag names may not begin with numbers or contain any of the following characters:

```
; @ # $ % ^ ( ) % + ? =
```

Thus, the tag `<$bisque>` is not acceptable. The colon (`:`) is a special symbol reserved to indicating a namespace. The dash (`-`) and dot (`.`) characters should be avoided.

Self-closing tags

Self-closing tags can be employed. An example is the empty tag:

```
<nothing_to_see_here/>
```

This compact structure is equivalent to the verbose structure:

```
<nothing_to_see_here> </nothing_to_see_here>
```

In the framework of the *xsl* programming language discussed in Chapter 3, a self-closing tag may serve to launch an application. In typesetting a document, a self-closing tag may force a line break or start a new chapter.

Self-closing tags are not necessarily devoid of information. For example, a self-closing tag may introduce an element described by attributes residing next to the tag name, as discussed in Section 2.2.1. An example is the tag:

```
<change_color new_color="moccasin"/>
```

where `new_color` is an attribute evaluated as *moccasin*.

Textual content

Text consisting of individual characters, words, and numbers can be inserted between an *xml* tag and its closure. Words and numbers broken into pieces by empty spaces lose their wholesome meaning and are treated as separate entities. For example, it is not appropriate to write:

```
<pi>3.14159 265358</pi>
```

The textual content of this element will be interpreted as a character string involving a blank space, not as a number. However, statements are allowed to extend over an arbitrary number of lines and the invisible character forcing a line break is inconsequential. A continuation mark at the end of a line is not required. For example, we may write:

```
<mytoolbox> currently, the toolbox is empty;
          please send in donations (especially hammers).</mytoolbox>
```

Exercise

2.1.1 *Self-closing tag*

Provide a sensible example of a self-closing tag involving an attribute.

2.2 Xml elements

Anything enclosed between a tag and its closure is an *xml* element, also called an element node. An element can be a physical object, an abstract object, a property of a parental object, or an instruction of a computer language that conforms with *xml* syntax and grammar. A self-closing tag is a vacant element.

2.2.1 Element attributes

An *xml* element may have attributes with arbitrary names conveying properties or descriptions, as illustrated in the following example:

```
<car color="red with black seats">
  ...
</car>
```

In this case, `color` is an attribute of a `car` evaluated by the character string `red with black seats`. The three dots denote additional content describing further element properties.

When employed, an *xml* attribute must be evaluated. For example, it is *not acceptable* to state:

```
✕ This is Wrong:

<car lemon>
  ...
</car>
```

The value of an attribute, whether a number or character string, must be enclosed by single or double quotes. For example, we may write:

```
<broom id="1" type= 'straw' color= "red"/>
  ...
</broom>
```

Element attributes can be used in self-closing tags, as discussed in Section 2.1. For example, we may write:

```
<force type="gravitational"/>
<force type="electromagnetic"/>
<force type="Coriolis"/>
<force type="centrifugal"/>
```

It is a good practice to avoid using attributes as much as possible and employ element nesting to describe object properties instead, as discussed in Section 2.2.2. One reason is that extracting attribute values requires more elaborate code. Another reason is that attributes cannot grow into data trees, and this may necessitate the restructuring of an *xml* document when additional information is supplied. This observation underlines the importance of proactive *xml* document design.

2.2.2 Property listing and nesting

Pairs of tags representing elements may be listed sequentially or otherwise nested multiple times to define a hierarchy of substructures. For example, we may write:

```
<car color="red with black seats">
  <make>Wartburg</make>
  <year>2007</year>
</car>
```

The word *Wartburg* should be regarded as the textual content, not the value, of the `make` element inside the `car` element. Similarly, we may write:

```
<car>
  <new>
    <make>Wartburg</make>
    <year>2007</year>
  </new>
</car>
```

The number *2007* should be regarded as the textual content, not the value, of the `year` element inside the `new` element.

Pairs of tags may not cross-over or overlap. Thus, the following structure is *not* acceptable:

✕ THIS IS WRONG:

```
<polynomial><orthogonal>Legendre</polynomial></orthogonal>
```

The correct structure is:

```
<polynomial><orthogonal>Legendre</orthogonal></polynomial>
```

Although *xml* tags can be arranged in a single line to save line breaks, this obscures the element structure.

The following *xml* element describes a triangle in terms of the coordinates of the three vertices in the *xy* plane specified as different *xml* children elements of the triangle:

```
<triangle>
  <x1>0.0</x1> <y1>0.0</y1>
  <x2>0.5</x2> <y2>0.3</y2>
  <x3>0.6</x2> <y3>-0.1</y3>
</triangle>
```

A person with elementary knowledge of geometry should be able to draw the triangle. If the order of the vertices is irrelevant and inconsequential, the triangle could be described as:

```
<triangle>
  <vertex> <x>0.0</x> <y>0.0</y> </vertex>
  <vertex> <x>0.5</x> <y>0.3</y> </vertex>
  <vertex> <x>0.6</x> <y>-0.1</y> </vertex>
</triangle>
```

This example illustrates that multiple vertex elements populating the same triangle element are allowed. To order the vertices, we may use an attribute:

```
<triangle>
  <vertex order="3'> <x>0.0</x> <y> 0.0</y> </vertex>
  <vertex order="1'> <x>0.5</x> <y> 0.3</y> </vertex>
  <vertex order="2'> <x>0.6</x> <y>-0.1</y> </vertex>
</triangle>
```

The `order` attribute allows us to assess whether the three vertices are arranged in the counterclockwise fashion in the *xy* plane by performing an appropriate geometrical test.

Talking out of turn

If we had misprinted the order of a vertex of a triangle so that two different vertices have the same order, the *xml* document would still be well-formed. This observation indicates that conforming with *xml* grammar does not guarantee contextual or mathematical sense.

Mixed content

Consider the following *xml* element:

```
<polynomial>
  Legendre
    <degree>23</degree>
</polynomial>
```

This element has mixed content consisting of (*a*) character data spelling the name *Legendre* and (*b*) one nested child element specifying the degree of the polynomial.

It is a good practice to avoid using mixed content as much as possible. When *xml* data are processed by an application written in the *xsl* language, mixed data are typically handled by templates playing the role of functions or subroutines, as discussed in Chapter 3.

2.2.3 Property and element tag names

Two different property tags may have the same name, provided that the corresponding elements are uniquely identified. For example, the following name scheme can be employed:

```
<country>

  <name>Transylvania</name>

  <famous_resident>
    <name>Dracula</name>
    <age>347</age>
  </famous_resident>

</country>
```

Note that the `name` tag appears twice with different meanings in this data structure. This is perfectly acceptable, for the context in which each name appears is clear.

White space

White space is generated by pressing the space bar, the Tab key, the Enter or Return key. *Xml* parsers are trained to retain white space inside an element, ignore white space between elements, and normalize white space by condensing it into a single space in attribute evaluations.

Exercises

2.2.1 *Record a truck and then a binomial*

(a) Record a truck as an *xml* element described by its make, year, color, and number of doors. (b) Record a binomial, $ax^2 + b\,x + c$, described by three possibly complex coefficients, a, b, and c.

2.2.2 *Mixed content*

Discuss a case where an *xml* element can be sensibly endowed with mixed content.

2.3 Comments

Comments are extremely helpful for providing explanations, documentation, and ancillary information in a data file or computer code. Comments can be inserted anywhere in an *xml* file according to the following format:

```
<!-- This was written on February 29, 2012 -->
```

or

```
<!-- The test of a first-rate intelligence is the ability
to hold two opposed ideas in the mind at the same time
and still retain the ability to function.

                       F. Scott Fitzgerald -->
```

Xml parsers are trained to ignore material between the comment delimiters `<!--` and `-->`. The *xml* comment convention is the same as that used in *html.* *Latex* accepts comments indicated by the percent mark (`%`) at the beginning of each commenting line or at any place in partially commenting line.

Two consecutive dashes (`--`) may not appear inside an *xml* comment.* For example, the following comment is not permissible:

```
<!-- use the Crank--Nicolson method -->
```

Although a comment may contain *xml* elements, an *xml* tag may not contain comments. Comments can be inserted inside the document type declaration DOCTYPE block defining a document type definition (*dtd*), as discussed in Section 2.10.

*A double dash encodes an *en dash* in *latex* typesetting, separating words that could each stand alone. In contrast, the hyphen separates words that convey meaning only as a pair. Thus, we must write: The Navier–Stokes equation is a second-order differential equation in space.

Commenting out blocks

The comment delimiters can be used to softly remove individual elements or groups of elements in an *xml* document. In the following example, delimiters are used to remove two zeros of a Bessel function:

```
<bessel_J0>
  <root>2.4048</root>

<!--
  <root>5.5201</root>   <root>8.6537</root>
-->

</bessel_J0>
```

Why comment out instead of remove? The discarded material may need to be temporarily disabled for a variety of reasons. Calmly think of a telemarketer removing a telephone number after receiving a complaint, only to reinstate it at a later time.

In scientific programming, commenting out lines is an invaluable method of debugging code. To comment out a line in *fortran*, we insert the `c` character at the beginning of the line. To comment out the whole or the tail end of a line in *fortran*, we put an exclamation mark (`!`) anywhere in the line. Text enclosed by the begin doublet `/*` and the end doublet `*/` is ignored in C or C++ code. To comment out the whole or tail end of a line in *Matlab*, we put a percent sign (`%`) anywhere in the line.

To comment out a block of text in a *latex* document, we use the *verbatim* package and wrap the disabled text inside the *comment* tag,

```
\begin{comment}
  ...
\end{comment}
```

where the three dots represent deactivated text.

Exercise

2.3.1 *Triple dash*

Can we put a triple dash (`---`) inside an *xml* comment?

2.4 Xml document declaration

The first line in an *xml* file declares that the file contains an *xml* document consistent with a specified *xml* version and possibly with a chosen character set. The minimal declaration stating consistency with the *xml* version 1.0 specification is:

```
<?xml version="1.0"?>
```

Version 1.1 extends the range of characters that can be employed.

Character sets

The *unicode* is a protocol mapping over one million characters, including letters, numbers, and other symbols, to a set of integers ranging from 0 to 1,114,112. Two mapping methods are available: the unicode transformation format (UTF) and the universal character set (UCS).

The UTF-8 and UTF-16 encodings are commonly employed.[*] UTF-8 maps characters to integers represented by 8 bits (one byte), whereas UTF-16 maps characters to integers represented by 16 bits (two bytes). The UTF-8 set is most compatible with the legacy American Standard Code for Information Interchange (*ascii*) mapping listed in Appendix A.

If western European languages are only used, the single byte ISO-8859-XX character set can be adopted, where `xx` is the version number. The ISO-8859-1 set is used by default in documents whose media type is *text*, such as those handled by a *web* server.

Character encoding and standalone specification

A typical *xml* declaration stating consistency with *xml* version 1.0 and the use of the ISO-8859-1 character set is:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

The *xml* data and possible processing instructions follow this declaration, as discussed later in this section. The most general *xml* document declaration has the typical form:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yo"?>
```

where `yo` can be `yes` or `no`. The optional `standalone` attribute is used when an internal document type definition (*dtd*) is employed, as discussed in Section 2.10.

It is worth emphasizing that the *xml* declaration becomes relevant only when an *xml* document is supplied for processing to an application. Particular applications may demand specific character sets.

[*] http://www.iana.org/assignments/charset-reg

Exercise

2.4.1 *ISO-8859-1*

List the first sixteen characters implemented in the Iso-8859-1 character set.

2.5 Character reference

Instead of typing a character, such as the pound sign (`#`), we may reference its *unicode*. For example, if the numerical code of a character in the decimal system is `78`, we may reference the character by entering:

```
&#78;
```

Notice the mandatory semi-colon (`;`) at the end. If the numerical code of a character in the hexadecimal system is `B6`, we may reference the character by entering:

```
&#xB6;
```

The hexadecimal encoding is indicated by the character `x`.

Predefined entities

Several predefined entities are available in *xml*. The *less than* (`<`) and *greater than* (`>`) signs, recognized as pointy or angle brackets and used as *xml* tag containers, can be referenced as:

```
&lt;    &gt;
```

The ampersand (`&`) character can be referenced as:

```
&amp;
```

For example, we may state:

```
<spice>
  salt &amp; pepper
</spice>
```

After the text enclosed by the spice tags has been processed *xml* processor, the following text will appear in the output: *salt & pepper*. Other predefined entities include the double quotation mark (") referenced as

```
&quot;
```

and the apostrophe or single quotation mark (') referenced as

```
&apos;
```

Exercise

2.5.1 *A word by character reference*

(*a*) Record the word *sanctimonious* by character reference to the Unicode. (*b*) Repeat for the word *promulgate.*

2.6 Language processing instructions

An instruction implemented in an appropriate language, such as *xsl*, *perl*, *python*, *java*, and others, or application, is called a processing instruction (Pi). To include a processing instruction in an *xml* document, we place it in a container opening with the pair `<?` and closing with the mirroring pair `?>`. A processing instruction becomes relevant only at the stage where an *xml* document is supplied to a processor for manipulation, bearing no relevance to the structure of the *xml* document.

Extensible stylesheet (xsl)

For example, an *xml* document may contain the following processing instruction whose meaning will be discussed in Chapter 3 in the context of the *xsl* processor:

```
<?  xml-stylesheet type="text/xsl" href="bilo.xsl" ?>
```

Briefly, this line invokes an extensible stylesheet (*xsl*) residing in a file whose name (*bilo.xsl*) is provided as a hypertext reference (*href*) attribute.

Cascading stylesheet (css)

Processing instructions are used routinely to link an *xml* or *html* document to a cascading stylesheet (*css*) by the typical statement:

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

A cascading stylesheet defines the global formatting of typesetting elements in an *xml* or *html* file. For example, a *css* may define the default display font, as discussed in Section 3.13.

Proprietary and other applications

Processing instructions can be used to convey information to a proprietary application, such as a word processor or a spreadsheet. A typical usage is:

```
<?  mso-application progid="Excel.Sheet" ?>
```

The name of the application (*Excel.Sheet*) is provided as a program id (*progid*) attribute.

A processing instruction can be used to insert comments in an *xml* document. However, better methods of inserting comments are available.

Processing instructions cannot be placed inside processing instructions, that is, they cannot be nested.

Xml declaration

In spite of its deceiving appearance, the *xml* declaration at the beginning of an *xml* document, such as

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

is *not* a processing instruction. The reason is that this declaration is completely understood by an *xml* parser. In contrast, a Pi can be understood only by an external application parsing the *xml* file.

Exercise

2.6.1 *Use of PIs*

Investigate the use of a Pi in an application of your choice.

2.7 Character data (Cdata)

Suppose that an *xml* document contains text that includes the following line:

```
<eigenvalue>0.302</eigenvalue>
```

which is to interpreted as verbatim text, as opposed to an *xml* element. Unfortunately, the string will be misconstrued as an *xml* element by the *xml* parser.

To prevent this misinterpretation, we may recast the line in terms of its character components as:

```
&lt; eigen &gt; 0.302 &lt; /eigenvalue&gt;
```

which eliminates the explicit presence of the troublesome pointy brackets (<>).

A more elegant and less confusing method involves putting the verbatim text inside the structure:

```
<![CDATA[
  ...
]]>
```

where the three dots indicate arbitrary text and Cdata stands for *character data* to be ignored by the *xml* parser. Note that the keyword Cdata must be capitalized. Also note the presence of two nested square brackets. In our example, we write:

```
<![CDATA[ <eigenvalue>0.302</eigenvalue> ]]>
```

In *web* programming applications, the character data structure is often used to enclose code.

It is understandable why unparsed character data (CDATA) may not contain the sequence:

```
]]>
```

The reason is that this pair will be falsely interpreted as the closing CDATA delimiter.

Deprecated tags allow us to insert verbatim text in an *html* document. Verbatim text in *latex*, such as `#$%(*&&`, can be placed inside the verbatim environment:

```
\begin{verbatim}
  #$%(*&&
\end{verbatim}
```

Verbatim text with small length can be placed inline using the typical *latex* structure:

```
\verb:this text appears verbatim:
```

where the semicolon (`:`) can be replaced by another character.

CDATA and PCDATA

To be precise, character data (CDATA) should be called unparsed character data, meaning that they are not parsed by an *xml* processor for the purpose of identifying elements and other structural information. In contrast, parsed character data are denoted as PCDATA.

Exercise

2.7.1 *Text explaining CDATA*

Is it possible to write a sentence discussing the CDATA statement in an *xml* document?

2.8 Xml root element

An *xml* file *must* contain a root element that encloses data and statements to be read by a person or processed by an application. Sometimes the root element is called the *document.*

The first tag in an *xml* file following the *xml* declaration and possible processing instructions defines the root element, and the last tag defines the closure of the root element. The typical structure of an *xml* document is:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<root_element_name>
  ...
</root_element_name>
```

where `root_element_name` can be any suitable name, and the three dots indicate additional data. The root element can be endowed with attributes. The root element of an *xhtml* document is `<html>`, and its closure is `</html>`. All elements inside the root element are children or descendants of the root element.

Self-closing root element

Strange though it may appear, a self-closing root element could be meaningful. For example, the following *xml* file may serve a purpose:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<shutdown what="system" when="now"/>
```

The name of the root element defined in the second line is `shutdown`. The mere presence of the root element is capable of triggering a specific type of action when the *xml* document is parsed to be processed by an application.

Doctype declaration

The name of the root element in an *xml* document can be stated explicitly in the preamble by the statement:

```
<!DOCTYPE root_element_name>
```

More generally, the Doctype declaration defines a data type definition (*dtd*), as discussed in Section 2.10.

Only one root element may be present in an xml file

Under no circumstances an *xml* file may have two root elements. Thus, the following structure is *not acceptable:*

```
✗ THIS IS WRONG:

<?xml version="1.0" encoding="ISO-8859-1"?>
<canoli>
  ...
</canoli>

<baklavas>
  ...
</baklavas>
```

One must choose either `canoli` or `baklavas`; it is wrong to indulge in both.

The root element of an xml data file is not a main program

Scientific computer programmers may be tempted to make a correspondence between the root element of the *xml* document containing data, the main program of a *fortran* code, or the main function of a C or C++ code. However, this correspondence is false. The sole similarity is that an *xml* data file may have one root element, and a C or C++ code may have only one main function.

In the *xml/xsl* framework discussed in Chapters 3 and 4, the root element of an *xml* document triggers the execution of the *xsl* code. The notion of data driving the execution is foreign to scientific programmers who are used to regarding data as optional companions of a standalone code.

The root element of an xml program file is not a main program

An *xml* file may contain code implementing computer language instructions, as discussed in Section 2.12. Even in these cases, the root element is not a main program or function, but only serves to introduce and set up the language processor, as discussed in Chapters 3 and 4 in the *xml/xsl* framework.

Exercise

2.8.1 *Unix root directory*

Discuss the analogy, if any, between the root directory of a *unix* system and the root element of an *xml* document.

2.9 Xml trees and nodes

Element nesting in an *xml* document results in an *xml* tree originating from the highest branching point called the *root*. Consider the following *xml* document:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<equation>
```

```
<algebraic>
  <quadratic>
    <first_coefficient> 1.5 </first_coefficient>
    <second_coefficient> 3.0 </first_coefficient>
    <third_coefficient> -4.0 </third_coefficient>
  </quadratic>
  <cubic>
    ...
  </cubic>
</algebraic>

<differential>
  ...
</differential>

<integral>
  ...
</integral>

</equation>
```

where three dots indicate additional lines of data. The name of the root element, enclosing all other children elements, is `equation`. Different types of equations are recorded in this document according to their classification.

The *xml* element structure forms a tree of nodes originating from the root, as depicted in Figure 2.1. Each labelled entry in this tree is an element node. Sibling, ascendant, and descendant nodes can be identified in an *xml* tree. In our example, algebraic, differential, and integral equations are siblings. Each node has one only one parent node. A leaf is a childless node.

Node list

Different quadratic algebraic equations defined in the document under discussion constitute a *node list* populating the same element node. Each quadratic algebraic equation could be attached to the appropriate branch near the south-western portion of the tree depicted in Figure 2.1, labeled by a numerical index starting at 0. Any *xml* element node can be populated with an arbitrary number of children nodes forming a node list parametrized by a numerical index.

Navigation path

It is important to emphasize that, even though the same element name may appear twice or multiple times at different branches of an *xml* tree, the corresponding nodes are distinguished by separate navigation paths. In our example, two such paths leading to an element named `second_order` are:

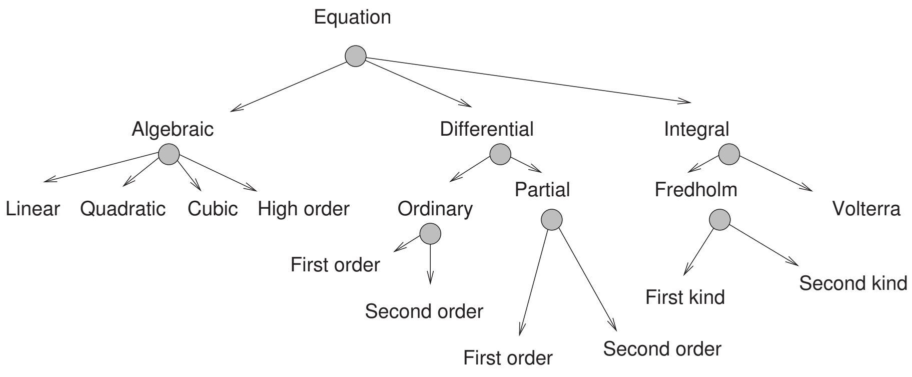


FIGURE 2.1 Tree structure of an *xml* document describing equations. The circles represent document nodes possibly serving multiple customers.

```
equation -> differential -> ordinary -> second_order
equation -> differential -> partial  -> second_order
```

These paths are analogous to library or executable paths of an operating system (Os). In an *xml* document, each path defines a unique *xml* node.

Relations and design

Relational context and possible future extensions must be understood for a successful data organization in an *xml* tree. Unless a large amount of disparate data are involved, common sense and a basic understanding of concepts and entities described in the *xml* document are the only prerequisites.

Basic rules

Two basic rules for an *xml* document to be well-formed may now be identified: (*a*) all tags must be properly nested and (*b*) only one root element playing the role of a main program may be present.

To illustrate further the meaning of proper nesting, we consider the following sentence: *I entered the barn, fed the donkey, exited the barn, and pet the donkey.* This sentence is not well-formed in the *xml* or any other rational framework. The proper structure is: *I entered the barn, fed the donkey, pet the donkey, and exited the barn.* In *fortran*, we can have an `If` loop inside a `Do` loop, but the `If` loop must close with an `End If` statement before the `Do` loop closes with an `End Do` statement.

Xml nodes

We have referred to the components of the tree shown in Figure 2.1 as element nodes. In fact, each identifiable component of an *xml* document is also an *xml* node. Examples include the root element (document), any other child or descendant element, an element attribute, a processing instruction (Pi), or a document type definition.

Exercise

2.9.1 *Xml tree*

Draw an *xml* tree containing square matrices in some rational taxonomy.

2.10 Document type definition and schema

We have seen that an *xml* document can be written using tags of our choice for clarity and easy reference. Eventually, the data will be read by a person or processed by a machine running an application. To prevent misinterpretation and ensure that the data are complete, an agreement on the tagging system describing element properties and attributes must be reached.

2.10.1 Internal document type definition (dtd)

In the simplest method, the agreement is implemented in a document type definition (*dtd*) that can be part of an *xml* document or accompany an *xml* document in an external file.

Consider the following well-formed *xml* document contained in the file *vehicles.xsl* of interest to a used-car dealer:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE inventory [
  <!ELEMENT inventory (car|truck)* >
  <!ELEMENT car (make,year) >
  <!ELEMENT truck (factory,built) >
  <!ELEMENT make (#PCDATA) >
  <!ELEMENT year (#PCDATA) >
  <!ELEMENT factory (#PCDATA) >
  <!ELEMENT built (#PCDATA) >
  <!ATTLIST car color CDATA "black" >
  <!ATTLIST truck color CDATA "black" >
  <!ENTITY found "We found a " >
]>

<inventory>

<car color="red">
  <make>Wartburg</make>
  <year>1966</year>
```

`|`	The vertical bar stands for logical OR to allow for a choice.
`+`	An element becomes mandatory by appending a plus sign to its name in the *dtd*.
`*`	An element is rendered optional by appending an asterisk to its name in the *dtd*.
`?`	An element is rendered optional and multivalued by appending a question mark to its name in the *dtd*.
CDATA	CDATA stands for unparsed character data.
`"`	The quotes enclose default attributes.
#PCDATA	#PCDATA stands for parsed character data.

TABLE 2.1 Conventions employed in defining elements and attributes in a document type definition (*dtd*).

```
</car>

<car color="green">
  <make>Yugo</make>
  <year>1970</year>
</car>

<truck color="white">
  <factory>Mercedes</factory>
  <built>1944</built>
</truck>

</inventory>
```

The name of the root element is `inventory`. Each car or truck is described by one attribute and two properties implemented as nested *xml* nodes.

DOCTYPE declaration

An internal *dtd* referring to the root element of the *xml* file and its descendants is implemented inside the DOCTYPE declaration, following the *xml* declaration in the first line of the *xml* document. Note that the keyword DOCTYPE is printed in upper-case letters. The *dtd* is implemented before the root element of the *xml* file.

The first entry in the *dtd* defines the root element of the *xml* file. Subsequent entries define elements by the keyword !ELEMENT and element attributes by the keyword !ATTLIST, subject to the conventions shown in Table 2.1.

In our example, the internal *dtd* specifies that the make and then the year of each car or truck must be declared, consistent with the *xml* data following the *dtd*. The tags `make` and `year` define two children elements of cars, whereas the tags `factory` and `built` define two children elements of trucks. These doublets convey similar information on the builder and date of built of each vehicle.

Unfortunately, a *dtd* does not allow us to assign the same name to children elements of two different elements. An *xml* schema definition (*xsd*) must be used when assigning the same name is desirable or necessary, as discussed in Section 2.10.3.

Element declaration

The most general element definition in a *dtd* is:

```
<!ELEMENT element_name element_content>
```

where `element_name` is the given element name. The `element_content` block defines the children elements, as shown in the *vehicles.xml* file. Additional examples are shown in Table 2.2(*a*). Other choices for `element_content` include Empty, Any, and a combination of parsed character data (#Pcdata) and children elements.

Element attribute declaration

The most general definition of an element attribute in a *dtd* is:

```
<!ATTLIST element_name attribute_name attribute_type attribute_value>
```

where `attribute_name` is the given attribute name describing the `element_name`.

The most common `attribute_type` is Cdata, as shown in the *vehicles.xml* file. Other choices, such as `ID`, are available. The `attribute_value` can be one of the following:

```
#REQUIRED      #IMPLIED      #FIXED "somevalue"     "somevalue"
```

The keyword Implied is used for an optional attribute that does not have a default value. The keyword Fixed is used for a mandatory attribute whose value cannot be changed in the *xml* document. Examples are shown in Table 2.2(*b*).

Entities

An *entity* can be defined in a *dtd* and then referenced in the *xml* file. For example, an entity named `showme` can be defined as:

```
<!ENTITY showme " The temperature measured at this point is:  ">
```

(*a*)

`<!ELEMENT inventory (car|truck)>`	One car or one truck is expected.
`<!ELEMENT inventory (car|truck)*>`	Any number of cars and trucks can be interspersed.
`<!ELEMENT inventory (car*|truck*)>`	Any number of cars *or* trucks are allowed; cars *and* trucks are not allowed.
`<!ELEMENT body (car*, truck*)>`	Any number of cars may be followed by any number of trucks.
`<!ELEMENT body (car, truck)*>`	Any number of ordered pairs of cars and trucks are allowed.
`<!ELEMENT car (make, year?)>`	The make of a car must be stated before the year, but the year is optional.

(*b*)

`<!ATTLIST computer model CDATA #REQUIRED>`	The computer model is required.
`<!ATTLIST student level CDATA "undergraduate">`	A default level is provided.
`<!ATTLIST bank branch CDATA #IMPLIED>`	The bank branch is optional.
`<!ATTLIST creature planet (earth|mars) "mars">`	A creature can be from earth or mars; the default planet is mars.
`<!ATTLIST customer planet CDATA #FIXED "earth">`	Every customer is from earth.

TABLE 2.2 An assortment of (*a*) element definitions and (*b*) element attribute definitions.

In the *xml* file, this entity is referenced as

```
&showme;
```

For example, we may state:

```
<temperature> &showme; 100.0 </temperature>
```

Summary

An internal *dtd* is placed immediately after the *xml* document declaration. The general statement of an internal *dtd* is:

```
<!DOCTYPE name_of_the_root_element [
  ...
]>
```

where the three dots denote statements that define elements, element attributes, entities, notation, processing instructions, comments, and references.

Validation

The process of inspecting an *xml* document against a *dtd* is called validation. Validation can be performed by opening an *xml* file with a *web* browser that is able to perform validation, or else by using an appropriate *xml* authoring tool. Online *xml* validators for internal and external *dtd*s accessible through the Internet are available.

A validator program called *xmllint* is available on a variety of platforms.* In our example, we open a terminal (command-line window) and type the line:

```
xmllint --valid --noout vehicles.xml
```

followed by the Enter keystroke. Nothing will be seen in the screen, indicating that the document has been validated against the internal *dtd*.

2.10.2 External document type definition (dtd)

A *dtd* can be placed in a separate file that accompanies an *xml* document. In that case, the internal *dtd* at the beginning of the *xml* file is replaced with the single line:

```
<!DOCTYPE inventory SYSTEM "DTD_file_name">
```

where `DTD_file_name` is the name of the file where the *dtd* is defined. The name of the *dtd* file is possibly preceded by a directory path, or else by a suitable

*http://xmlsoft.org

web address identified as a uniform resource locator (*url*). In the last case, the keyword SYSTEM in the *dtd* declaration can be replaced by the keyword PUBLIC. The external *dtd* file itself contains the text enclosed by the square brackets of an internal *dtd*.

If the PUBLIC keyword is used, a formal public identifier (*fpi*) must be used in the DOCTYPE declaration. The *fpi* consists of four fields separated by a double slash. For example, the DOCBOOK *dtd* discussed in Section 1.4.1 is invoked by the following statement where the *fpi* is printed in the second line:

```
<!DOCTYPE article PUBLIC
  "-//OASIS//DTD DocBook XML V4.1.2//EN"
    "http://www.oasis-open.org/docbook/xml/4.1.2/docbookx.dtd">
```

The dash in the first field of this *fpi* indicates that the *dtd* has not been approved by a recognized authority; the second field indicates that the organization OASIS is responsible for this *dtd*; the third field contains additional information; the fourth field is an English language specification (EN).

If an external *dtd* is used, the *xml* document declaration must specify that the *xml* document is not standalone. For example we may declare:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
```

If the `standalone` attribute is omitted, the default state is affirmative.

A graph

In mathematics, a graph is a set of nodes (vertices) connected by edges (links). Vertices and edges are assigned arbitrary and independent numerical labels. As an example, we consider the following file named *graph.xml* containing information on three vertices and two edges of a graph:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>

<!DOCTYPE network SYSTEM "graph.dtd">

<network>

<vertex id="1">
  <adjacent>2</adjacent>
  <adjacent>3</adjacent>
</vertex>

<vertex id="2">
  <adjacent>3</adjacent>
  <adjacent>1</adjacent>
</vertex>
```

```
<vertex id="3">
  <adjacent>1</adjacent>
  <adjacent>2</adjacent>
</vertex>

<edge id="1">
  <incidence1>1</incidence1>
  <incidence2>2</incidence2>
</edge>

<edge id="2">
  <incidence1>2</incidence1>
  <incidence2>3</incidence2>
</edge>

</network>
```

The name of the root element is `network`. The label of each vertex or edge is recorded as an attribute named `id`. The vertices adjacent to each vertex are recorded along with two incidence indices for each edge specifying the labels of the first and second end points of each edge. Cursory inspection reveals that this is a `V`-shaped graph consisting of three vertices connected by two edges.

The accompanying external *dtd* file named *graph.dtd* referenced in the second line reads:

```
<!ELEMENT network (vertex*,edge*) >
<!ELEMENT vertex (adjacent*) >
<!ELEMENT adjacent (#PCDATA) >
<!ELEMENT edge(incidence1, incidence2) >
<!ELEMENT incidence1 (#PCDATA) >
<!ELEMENT incidence2 (#PCDATA) >
<!ATTLIST vertex id CDATA #REQUIRED >
<!ATTLIST edge id CDATA #REQUIRED >
```

Assuming that the *graph.xml* and *graph.dtd* files reside in the same directory (folder), we may validate the *xml* data using the *xmllint* application by opening a terminal (command-line window) and issuing the command:

```
xmllint -noout graph.xml --dtdvalid graph.dtd
```

Nothing will appear on the screen, indicating that the *xml* file has been validated against the external *dtd*.

Combining an internal with an external dtd

An *xml* document can have an internal *dtd*, an external *dtd*, or both. In our example, we may use the declaration:

```
<!DOCTYPE network SYSTEM "graph.dtd" [
  <!ENTITY found "Number of nodes:  " >
]>
```

which adds an entity to the external *dtd*. The same element cannot be defined both in the internal and external *dtd*.

2.10.3 Xml schema definition (xsd)

Like a *dtd*, an *xml* schema definition (*xsd*) determines the required structure of an *xml* document.* The word schema (pl. schemata) should not be confused with the possibly pejorative scheme.

Unlike a *dtd*, an *xsd* may incorporate advanced features that allow us to define data types, such as integers, real numbers, and character strings, and also employ namespaces, as discussed in Section 2.11. The use of a *xsd* is recommended over a *dtd* in advanced and commercial applications.

An *xml* schema definition is contained in a file identified by the suffix `.xsd`. An interesting feature of an *xsd* is that its implementation follows *xml* grammar. As an example, we consider the data contained in the following *xml* file named *pets.xml*:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<pets xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="pets.xsd">

    <dog>Pluto</dog>
    <cat>Garfield</cat>

</pets>
```

The second line makes reference to a schema contained in the following file named *pets.xsd*:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="pets">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="dog" type="xs:string" />
        <xs:element name="cat" type="xs:string" />
      </xs:sequence>
```

*$\Sigma\chi\eta\mu\alpha$ is a Greek word meaning shape, form, layout, or framework.

```
    </xs:complexType>
  </xs:element>

</xs:schema>
```

Assuming that the files *graph.xml* and *graph.xsd* reside in the same directory (folder), we may validate the *xml* data using the *xmllint* application by opening a terminal (command-line window) and issuing the command:

```
xmllint -noout pets.xml --schema pets.xsd
```

Nothing will appear on the screen, indicating that the *xml* file has been validated against the *xsd*.

2.10.4 Loss of freedom

It is clear that, by using a *dtd* or *xsd*, we give up our freedom to arbitrarily but sensibly define element tags, attributes, and nodes in an *xml* document. This is certainly disappointing, as lamented on previous occasions.

However, it must be emphasized that the use of a *dtd* or *xsd* is optional and relevant only at the stage where the *xml* data will be communicated or retrieved. In professional applications, *xml* authors are happy to conform with universal standards designed by others, so that their documents can be smoothly processed.

Exercise

2.10.1 *Dtd for a polynomial*

Write a *dtd* pertaining to the real, imaginary, or complex roots of an Nth-degree polynomial.

2.11 Xml namespaces

The finite-element and boundary-element methods are advanced numerical methods for solving differential equation in domains with arbitrary geometry.[*] The main advantage of the boundary-element method is that only the boundary of a given solution domain needs to be discretized into line elements in two dimensions or surface elements in three dimensions. The finite-element method is able to tackle a broader class of differential equations, albeit at a significantly elevated cost.

[*]Pozrikidis, C. (2008) *Numerical Computation in Science and Engineering*, Second Edition, Oxford University Press.

In both the finite-element and boundary-element method, geometrical elements with different shapes and sizes can be employed to accommodate the geometry of the solution domain. Examples include straight segments, circular arcs, triangles, and rectangles.

Finite-element and boundary-element grids

Assume that an *xml* document contains information on finite and boundary elements comprising corresponding grids used to solve the Laplace equation. It is desirable to use the name *element* in both cases, albeit in different contexts.

To achieve this, we introduce two *xml* namespaces (*xmlns*), one for the finite elements and the second for boundary elements. The content of the pertinent *xml* file named *elements.xml* is:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<laplace>

  <fem:grid xmlns:fem="femuri">
    <fem:element id="1" shape="triangular" nodes="3"/>
    <fem:element id="2" shape="rectangular" nodes="4"/>
  </fem:grid>

  <bem:grid xmlns:bem="bemuri">
    <bem:element id="1" shape="linear" nodes="2"/>
    <bem:element id="2" shape="circular" nodes="3"/>
  </bem:grid>

</laplace>
```

The name of the root element is `laplace`. Two uniform resource identifiers (*uri*) identified with two uniform resource names (*urn*), arbitrarily called *femuri* and *bemuri*, are used in this document. The *uris* are used to evaluate the *xmlns* attribute of the corresponding element, *xmlns:bem* and *xmlns:fem*. *Web* addresses called uniform resource locators (*url*) where information on each namespace is given are used in most applications as *uris*.

It is important to note that, if we had discarded the prefixes `fem:` and `bem:` in the *xml* file, we would no longer be able to distinguish between the finite- and boundary-element grids and identify the corresponding elements employed.

Alternatively, namespaces can be defined and evaluated as attributes of the root element of an *xml* document, as shown in the following file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<laplace xmlns:fem="femuri" xmlns:bem="bemuri">

<fem:grid>
```

```
  <fem:element id="1" shape="triangular" nodes="3"/>
  <fem:element id="2" shape="rectangular" nodes="4"/>
</fem:grid>

<bem:grid>
  <bem:element id="1" shape="linear" nodes="2"/>
  <bem:element id="2" shape="circular" nodes="3"/>
</bem:grid>

</laplace>
```

The name of the root element is `laplace`.

In professional applications, namespaces are helpful when different parts of a code are written by different software engineering teams. Each team can be identified by its own namespace for credit or blame.

Default namespace

To simplify the notation, we may introduce a default namespace that lacks a prefix. Only one default namespace is allowed in an *xml* document. For example, a default finite-difference grid can be introduced by the following lines:

```
<grid xmlns="someuri">
  <xsize>16</xsize>
  <ysize>32</ysize>
</grid">
```

The finite-element namespace is the default namespace in the following *xml* document:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<laplace xmlns="femuri" xmlns:bem="bemuri">

  <grid>
    <element id="1" shape="triangular" nodes="3"/>
    <element id="2" shape="rectangular" nodes="4"/>
  </fem:grid>

  <bem:grid>
    <bem:element id="1" shape="linear" nodes="2"/>
    <bem:element id="2" shape="circular" nodes="3"/>
  </bem:grid>

</laplace>
```

Unless specified otherwise, all descendants of an element in the default namespace also fall in the default namespace.

Qualified names

An *xml* qualified name (*qname*) is an *xml* element name optionally preceded by a namespace, called the prefix, and a colon (`:`). An example is `bem:element`.

Exercise

2.11.1 *Namespaces*

Discuss a scientific or engineering application where the use of two namespaces is desirable.

2.12 Xml formatting of computer language instructions

A conditional block in a computer code has the following generic pseudocode structure:

```
If (something_is_true) then
  ...
End If
```

where the three lines represent instructions to be followed only if the statement `something_is_true` is true. A possible recasting of this block into *xml* compliant form is:

```
<if test="something_is_true">
  ...
</if>
```

The direct translation is possible because both the conditional block of the pseudocode and the `if` element of the *xml* document require closure. More generally, the syntax of any suitable computer language could be restated to comply with *xml* conventions.

As an example, we consider the following complete *fortran* program that defines and adds two numbers, and then prints their sum on the screen:

```
      program vasvas

      a = 5.87
      b = 6.01
      c = a+b
      print c

      stop
      end
```

Note that six mandatory blank spaces have been inserted at the beginning of each line. The `stop` statement refers to execution, and the `end` statement refers to compilation.

A possible equivalent code written in *xml* is:

```
<?xml version="1.0"?>

<fortran:program xmlns:fortran="primm">
  <fortran:variable type="real" name="a" value="5.87"/>
  <fortran:variable type="real" name="b" value="6.01"/>
  <fortran:variable type="real" name="c" value="a+b"/>
  <fortran:print format="any" name="c"/>
<fortran:program>
```

A program can be written that translates this *xml* document into *fortran* code, and *vice versa*. Although a complete set of semantics is conveyed by the *xml* document, the visual cluttering and the repetition of terms are distracting.

Xml programming language implementation

Programming languages that employ *xml* grammar and syntax are available. An example is the extensible stylesheet language (*xsl*) discussed in Chapters 3 and 4. A comprehensive list of other languages is available on the Internet.[*]

In using an *xml* compliant language, two *xml* files are necessary: an *xml* data file containing input, and an *xml* program file implementing language instructions. Each file has its own root element serving a different purpose. Conversely, an arbitrary *xml* file can contain either data or computer programming instructions.

Xsl

The typical structure of an *xsl* program file is:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  ...
</xsl:stylesheet>
```

where the three dots indicate additional lines of code, as discussed in Chapters 3 and 4. The first line is an *xml* declaration. The second line introduces the root element named `xsl:stylesheet` and defines the *xsl* namespace. Other namespaces could be added to this line, if necessary. Since *xsl* statements adhere to *xml* standards, the first line of the code is not mandatory.

[*] http://en.wikipedia.org/wiki/List_of_XML_markup_languages

Scalable vector graphics (svg)

The scalable vector graphics (*svg*) suite defines a family of *xml* tags that describe geometrical elements representing two-dimensional vector graphics. *Svg* processors are embedded in *web* browsers and other graphics applications.

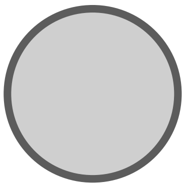
As an example, we consider the following *svg* file written in the *xml* format:

```
<?xml version="1.0" standalone="no"?>

<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg xmlns="http://www.w3.org/2000/svg" version="1.1">
  <circle cx="64" cy="64" r="32" stroke="red"
    stroke-width="2" fill="yellow" />
</svg>
```

Because an external *dtd* is employed, this *xml* file is not standalone. The name of the root element, qualified by two attributes, is `svg`. One *svg* element implementing a yellow disk enclosed by a red circle resides inside the root element. Opening the *xml* file with a *web* browser produces the following display:



Svg and *mathml* have an element named *set*. Different namespaces must be used when *svg* and *mathml* are simultaneously employed.

tikzpicture

In typesetting this book, the previous display of the disk was programmed using the *tikzpicture latex* package, which also produces scalable vector graphics. The typesetting instructions in the *latex* document are:

```
 \begin{centering}
     \begin{tikzpicture}
          \draw[fill=yellow] ellipse (32pt and 32pt);
          \draw[line width=1mm,color=red] circle (32pt);
     \end{tikzpicture}
\end{centering}
```

A variety of other *svg* graphics elements and corresponding *tikzpicture* elements are available.

Exercises

2.12.1 *Your computer language in xml*

Recast a scientific code in a language of your choice into *xml* format.

2.12.2 *Aragorn in xml*

Write a code of your choice in a fictitious computer language called *aragorn* according to *xml* conventions.