

XML: Looking at the Forest Instead of the Trees

**Guy Lapalme
Professor**

**Département d'informatique et de recherche opérationnelle
Université de Montréal**

**C.P. 6128, Succ. Centre-Ville
Montréal, Québec
Canada H3C 3J7
lapalme@iro.umontreal.ca**

<http://www.iro.umontreal.ca/~lapalme/ForestInsteadOfTheTrees/>



Publication date August 01, 2022

XML: Looking at the Forest Instead of the Trees

Guy Lapalme

Professor

Département d'informatique et de recherche opérationnelle Université de Montréal

C.P. 6128, Succ. Centre-Ville
Montréal, Québec
Canada H3C 3J7
lapalme@iro.umontreal.ca

<http://www.iro.umontreal.ca/~lapalme/ForestInsteadOfTheTrees/>

Publication date August 01, 2022

Abstract

This tutorial gives a high-level overview of the main principles underlying some XML technologies: DTD, XML Schema, RELAX NG, Schematron, XPath, XSL stylesheets, Formatting Objects, DOM, SAX and StAX models of processing. They are presented from the point of view of the computer scientist, without the *hype* too often associated with them. We do not give a detailed description but we focus on the relations between the main ideas of XML and other computer language technologies. A single *compact pretty-print example* is used throughout the text to illustrate the processing of an XML structure with XML technologies or with Java programs. We also show how to create an XML document by programming in Java, in Ruby, in Python, in PHP, in JavaScript and in Swift.

The source code of the example XML files and the programs are available either at the companion web site of this document or by clicking on the file name within brackets at the start of the caption of each example. The source file will then be shown in a web page or in a new tab of the browser. Should the page be interpreted by the browser, the reader can ask for the source of the page.

Acknowledgements

The writing of this XML tutorial started in the Fall of 2002 during my sabbatical at the Université de Grenoble and at Xerox Research Centre Europe. I wish to thank Gilles Sérasset, Christian Boitet, Pierre Isabelle and Marc Dymetman for many fruitful discussions.

Since then, the document has been improved (at least it has increased in the number of pages...) after having used it in teaching undergraduate and graduate courses at the Université de Montréal: IFT3225 and IFT6281. I especially thank Fabrizio Gotti for his careful proofreading and many insightful comments.

In 2007, I decided to start practicing what I preach by using XML technologies for the organization of the manuscript of this tutorial. Fabrizio Gotti converted the original LaTeX files to DocBook so that now PDF and HTML versions can be produced from a single set of XML source files.

Table of Contents

1. Introduction	1
2. Instance Document	9
2.1. Namespaces	13
3. Document Validation	17
3.1. Document Type Definition (DTD)	17
3.1.1. Associating an Instance File with a DTD	21
3.2. Schema	22
3.2.1. Simple Types	33
3.2.2. Complex Types	34
3.2.3. Namespaces in Schemas	34
3.2.4. Overview of the XML Schemas	35
3.3. RELAX NG	38
3.4. Schematron	43
3.5. Associating an Instance File with a Schema	48
3.6. Additional Information on XML Schema	49
4. XPath	51
4.1. XPath expression components	51
4.2. XPath functions	55
4.3. XPath examples	57
4.4. Additional Information on XPath	58
5. Document Transformation	59
5.1. XSL Transformations	60
5.2. Transformation in HTML	63
5.2.1. Table	63
5.2.2. Computing New Information	68
5.2.3. Bulleted Lists	76
5.3. Transformation into a Compact Textual Form	80
5.4. Transformation into PDF with XSL-FO	84
5.4.1. XSL-FO Input to the Renderer	86
5.4.2. From the Instance Document to the XSL-FO file	87
5.5. Transformation with a CSS	93
5.6. Associating an Instance File to a Stylesheet	95
5.7. Additional Information on XSL	97
6. Document Query	99
6.1. XQuery output in HTML	101
6.1.1. Table	101
6.1.2. Computing New Information	103
6.1.3. Bulleted Lists	107
6.2. Transformation into a Compact Textual Form with XQuery	110
6.3. Querying an instance file	112
6.4. Additional Information on XQuery	112
7. RDF : Resource Description Framework	113
7.1. Triples in RDF/XML	116
7.2. RDF Schema	123
7.3. RDF queries	124
7.4. RDF versus XML	125

7.5. Additional Information on RDF	126
8. Document Processing by Programming in Java	127
8.1. Document Object Model (DOM)	128
8.2. Simple API for XML (SAX)	132
8.3. Stream API for XML (StAX)	136
8.4. Showing an Interactive Tree View	138
8.4.1. Building a JTree with DOM	139
8.4.2. Building a JTree with SAX	141
8.4.3. Building a JTree with StAX	143
8.5. Additional Information on Programming Models	144
9. Document Creation by Programming in Java	145
9.1. Creating a DOM Document	145
9.2. Creating a Document with SAX Events	149
9.3. Creating a Document with StAX streaming	153
9.4. Additional Information on XML Document Creation	155
10. Alternative approaches	157
10.1. XML processing with Ruby	157
10.1.1. DOM parsing using Ruby	158
10.1.2. SAX parsing using Ruby	159
10.1.3. Creating an XML document using Ruby	161
10.2. XML processing with Python	165
10.2.1. DOM parsing using Python	165
10.2.2. SAX parsing using Python	166
10.2.3. StAX parsing using Python	168
10.2.4. Creating an XML document using Python	171
10.2.5. Other means of dealing with XML documents using Python	173
10.3. XML processing with PHP	178
10.3.1. DOM parsing using PHP	178
10.3.2. SAX parsing using PHP	180
10.3.3. StAX parsing using PHP	183
10.3.4. Creating an XML document using PHP	184
10.3.5. Other means of dealing with XML documents using PHP	187
10.4. XML processing with JavaScript	193
10.4.1. DOM parsing using JavaScript	194
10.4.2. Creating an XML document using JavaScript	196
10.5. XML processing with Swift	199
10.5.1. DOM parsing using Swift	199
10.5.2. SAX parsing using Swift	200
10.5.3. Creating an XML document using Swift	203
10.6. XML processing with E4X	206
10.6.1. DOM parsing using E4X	209
10.6.2. Creating an XML document using E4X	210
10.7. XML alternative notations	214
10.7.1. JSON	214
10.7.2. YAML	220
10.8. Additional information on alternative approaches	222
11. Conclusion	223
Bibliography	225
A. Some XML Related Technologies and Systems	229

B. Quick Reference Tables	231
B.1. Regular expression	231
B.2. DTD	231
B.3. XML Schema	232
B.4. RELAX NG	233
B.5. Schematron	234
B.6. XSLT	235
C. XML Production of this Document	237
D. Instance documents	243
Index	249

List of Figures

1.1. An XML structure and the corresponding tree [Wine.xml]	1
1.2. Web browser, summary and grid editor views of an XML file	3
1.3. Relations between some XML technologies	4
1.4. HTML source of the compact form and its rendering in a browser.	5
1.5. Compact forms in text and in PDF	6
3.1. Graphical view of the schema for the cellar book	25
3.2. Graphical view of the schema for the wine catalog	29
3.3. Built-in datatypes for XML Schema	33
4.1. Some XPath axes	53
5.1. Web browser rendering of Example 5.1 produced by running Example 5.2 on Example 2.3	64
5.2. HTML rendering of Example 5.3	69
5.3. HTML rendering of Example 5.6	76
5.4. Three pages of PDF output of compaction by Formatting Objects	85
5.5. Outline of the XSL-FO file produced by running Example 5.10 on Example 2.2	86
5.6. CSS formatting of Example 2.2	95
7.1. Semantic Stack	114
7.2. Selected information from the cellar-book	117
8.1. JTree display of Example 2.2	139
10.1. Display of the webpage for exercising JavaScript compaction and expansion.	193
10.2. HTML for JavaScript compact and expand	194
10.3. E4X expression examples	208
C.1. Overview of organization of the DocBook XML source files of this document	238

List of Tables

3.1. DTD syntax reminder	18
3.2. XML Schema syntax reminder	23
3.3. RELAX NG Compact syntax reminder	39
3.4. Schematron syntax reminder	45
4.1. A selection of XPath functions	55
5.1. XSLT syntax reminder	62
7.1. RDF Triples	116
7.2. Comparison of Turtle brackets with triples	121
B.1. Regular expression syntax	231

List of Examples

2.1. Outline of <code>CellarBook.xml</code> including <code>WineCatalog.xml</code> in a different namespace	9
2.2. Excerpt of <code>CellarBook.xml</code> , the XML instance document holding the content of the cellar	10
2.3. Excerpt of <code>WineCatalog.xml</code>	12
2.4. [<code>NamespaceExample.xml</code>] A simplistic example of declaration and use of namespaces	13
3.1. [<code>CellarBook.dtd</code>] DTD for the cellar book, validation of the instance file in Example 2.2	19
3.2. [<code>WineCatalog.dtd</code>] DTD to validate the wine catalog, included in Example 3.1	20
3.3. [<code>CellarBook.xsd</code>] XML Schema for the cellar book, validation of the instance file in Example 2.2.	25
3.4. [<code>WineCatalog.xsd</code>] XML Schema file to validate the instance document shown in Example 2.3.	29
3.5. Outline of <code>CellarBook.xsd</code> importing Example 3.4 in a different namespace.	36
3.6. [<code>CellarBook.rnc</code>] RELAX NG Compact schema for the cellar book to validate Example 2.2.	39
3.7. [<code>WineCatalog.rnc</code>] RELAX NG Compact schema for the wine catalog to validate Example 2.3	41
3.8. [<code>CellarBook.sch</code>] ISO-Schematron for the cellar book to validate Example 2.2.	45
4.1. XPath expression examples	57
5.1. HTML tabular output of the red wines of the catalog produced by Example 5.2	64
5.2. [<code>WineCatalog.xsl</code>] XSLT stylesheet to select the red wines in the catalog	66
5.3. HTML output by the XSLT code is shown in Example 5.4	69
5.4. [<code>CellarBook.xsl</code>] XSLT stylesheet to produce information about the cellar	71
5.5. [<code>built-in-template-rules.xsl</code>] Built-in template rules for XSLT.	75
5.6. Extract of the HTML output produced by the transformation of Example 5.4 on Example 2.2	76
5.7. [<code>compactHTML.xsl</code>] XSLT transformation to produce a bulleted list from the cellar book	78
5.8. Text compaction of the cellar book of Example 2.2	80
5.9. [<code>compact.xsl</code>]: Stylesheet used to transform Example 2.2 into Example 5.8	82
5.10. [<code>compactFO.xsl</code>] Transformation of the cellar book into a colored nested blocks representation	89
5.11. [<code>compact.css</code>] for displaying the content of the cellar book	93
6.1. [<code>WineCatalog.xq</code>] XQuery script to select the red wines in the catalog	101
6.2. [<code>CellarBook.xq</code>] XQuery script to produce information about the cellar	104
6.3. [<code>compactHTML.xq</code>] XQuery transformation to produce a bulleted list from the cellar book	108
6.4. [<code>compact.xq</code>]: Stylesheet used to transform Example 2.2 into Example 5.8	111
7.1. [<code>CBWC-RDF-S.rdf</code>] Subject, Predicate and Object triples for the cellar book in RDF/XML.	118
7.2. [<code>CBWC-RDF-S.ttl</code>] The Turtle version of Example 7.1	121
7.3. [<code>CBWC-RDF-S.rq</code>] SPARQL queries on Example 7.2	125
8.1. [<code>DOMCompact.java</code>] Text compaction of the cellar book with Java using the DOM model	128
8.2. [<code>CompactErrorHandler.java</code>] Error handler of the DOM parsing of Example 8.1	131
8.3. [<code>SAXCompact.java</code>] Text compaction of the cellar book with Java using the SAX model	133
8.4. [<code>CompactHandler.java</code>] SAX handler for text compacting an XML file such as Example 2.2	134
8.5. [<code>StAXCompact.java</code>] Text compaction of the cellar book with Java using the SAX model	137
8.6. [<code>TreeViewer.java</code>]: JTree building with DOM and StAX Processing of an XML file	140
8.7. [<code>JTreeHandler.java</code>]: JTree building with SAX Processing of an XML file	142

9.1. [CompactTokenizer.java]: Specialized stream tokenizer that ignores blank tokens	146
9.2. [DOMExpand.java]: Compact form parsing to create a DOM XML document	147
9.3. [SAXExpand.java]: XML document creation using SAX events	149
9.4. [CompactReader.java]: Compact form parsing to generate SAX events	150
9.5. [StAXExpand.java]: XML document creation using the StAX approach	153
10.1. [DOMCompact.rb] Text compaction of the cellar book with ruby using the DOM model	158
10.2. [SAXCompact.rb] Text compaction of the cellar book with ruby using the SAX model	159
10.3. [CompactHandler.rb] Ruby SAX handler for text compacting Example 2.2	159
10.4. [CompactTokenizer.rb] Specialized string scanner that returns tokens of compact form.	161
10.5. [DOMExpand.rb] Ruby compact form parsing to create an XML document	162
10.6. [DOMCompact.py] Text compaction of the cellar book with Python using the DOM model	165
10.7. [SAXCompact.py] Text compaction of the cellar book with python using the SAX model	166
10.8. [CompactHandler.py] Python SAX handler for text compacting Example 2.2	167
10.9. [XMLStreamReader.py] Text compaction of the cellar book with Python using the SAX model	168
10.10. [StAXCompact.py] Text compaction of the cellar book with Python using the StAX mod- el	170
10.11. [CompactTokenizer.py] Specialized string scanner that returns tokens of compact form.	171
10.12. [DOMExpand.py] Python compact form parsing to create an XML document	172
10.13. [ETCompact.py] Python compaction of an XML file using ElementTree nodes.	174
10.14. [ETExpand.py] Python compact form parsing to create a ElementTree document	175
10.15. [DOMCompact.php] Text compaction of the cellar book with php using the DOM model	178
10.16. [compactHTML.php] HTML compaction of the cellar book with PHP using the DOM mod- el	179
10.17. [SAXCompact.php] Text compaction of the cellar book with php using the SAX model	181
10.18. [CompactHandler.php] PHP SAX handler for text compacting Example 2.2	181
10.19. [StAXCompact.php] Text compaction of the cellar book with PHP using the SAX mod- el	183
10.20. [CompactTokenizer.php] Specialized string scanner that returns tokens of compact form.	184
10.21. [DOMExpand.php] PHP compact form parsing to create an XML document	186
10.22. [XSLcompact.php] PHP compaction of an XML file using an XSLT stylesheet.	187
10.23. [SimpleXMLPath.php] SimpleXML file loading followed by PHP SimpleXML expres- sions	188
10.24. [SimpleXMLCompact.php] PHP compaction of an XML file using a SimpleXML.	189
10.25. [SimpleXMLExpand.php] PHP compact form parsing to create a SimpleXMLElement document	191
10.26. [DOMCompact.js] Text compaction of the cellar book with JavaScript using the DOM model	194
10.27. [CompactTokenizer.js] JavaScript specialized string scanner that returns tokens of compact form	196
10.28. [DOMExpand.js] JavaScript compact form parsing in order to create an XML document	197
10.29. [DOMCompact.swift] Text compaction of the cellar book with Swift using the DOM mod- el	199
10.30. [SAXCompact.swift] Text compaction of the cellar book with swift using the SAX mod- el	201

10.31. [DOMExpand.swift]	Swift compact form parsing to create an XML document	203
10.32. [DOMCompact.as]	Text compaction of the cellar book with E4X using the DOM model	209
10.33. [CompactTokenizer.as]	E4X specialized string scanner that returns tokens of compact form	210
10.34. [DOMExpand.as]	E4X compact form parsing in order to create an XML document	212
10.35. [WineList.xml]	A small wine list in XML	215
10.36. [WineList.json]	A small wine list in JSON	215
10.37. [xml2json.xsl]	XSLT stylesheet to convert an XML file into JSON.	216
10.38. [WineList.yaml]	YAML version of Example 10.35	220
C.1. [DBTemplates.xml]	Examples of the most common uses of DocBook 5 elements in this document.	239
D.1.	XML content of the file describing the cellar book	243
D.2.	XML content of the file describing the wine catalog	246

Acknowledgements

The writing of this XML tutorial started in the Fall of 2002 during my sabbatical at the Université de Grenoble and at Xerox Research Centre Europe. I wish to thank Gilles Sérasset, Christian Boitet, Pierre Isabelle and Marc Dymetman for many fruitful discussions.

Since then, the document has been improved (at least it has increased in the number of pages...) after having used it in teaching undergraduate and graduate courses at the Université de Montréal: IFT3225 and IFT6281. I especially thank Fabrizio Gotti for his careful proofreading and many insightful comments.

In 2007, I decided to start practicing what I preach by using XML technologies for the organization of the manuscript of this tutorial. Fabrizio Gotti converted the original LaTeX files to DocBook so that now PDF and HTML versions can be produced from a single set of XML source files.

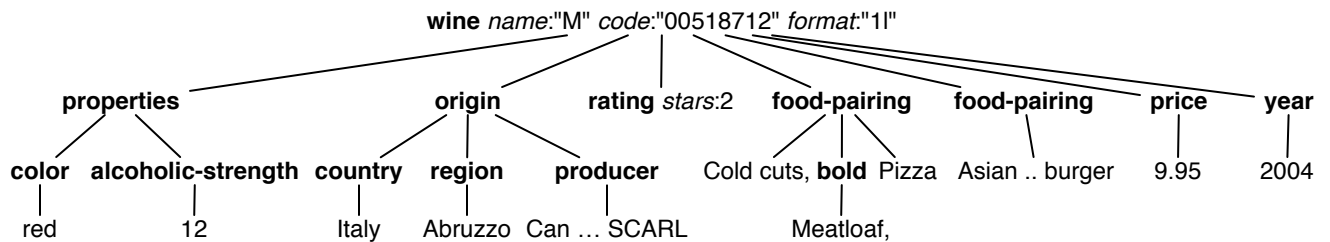
Chapter 1. Introduction

XML has been developed to facilitate the annotation of information to be shared between computer systems. Because it is intended to be easily generated and parsed by computer systems on diverse platforms, its format is based on character streams rather than internal binary ones. Being character based, it also has the nice property of being readable and editable by humans using standard text editors.

Figure 1.1. An XML structure and the corresponding tree [wine.xml]

A simple XML structure (top) and an equivalent tree structure in which the element names are shown in bold and the attribute names in italics. In the tree, the *real* information is the character data which appears in roman font. This shows the relations between nodes: *properties* has *wine* as *parent* and *color*, *alcoholic-strength* as children; a sibling of *region* is *country*.

```
<?xml version="1.0" encoding="UTF-8"?>
<wine name="M" code="00518712" format="11"> <!-- code is the same as for SAQ.com-->
  <properties>
    <color>red</color>
    <alcoholic-strength>12</alcoholic-strength>
  </properties>
  <origin>
    <country>Italy</country>
    <region>Abruzzo</region>
    <producer>Cantina Miglianico SCARL</producer>
  </origin>
  <rating stars="2"/>
  <food-pairing>Cold cuts, <b>Meatloaf,</b> Pizza</food-pairing>
  <food-pairing>Asian spicy pork burger</food-pairing>
  <price>9.95</price>
  <year>2004</year>
</wine>
```



XML is based on a uniform, simple and yet powerful model of data organization: the generalized tree. Such a tree is defined as either a single element or an element having other trees as its sub-elements called *children* (see middle of Figure 1.1). This is the same model as the one chosen for the Lisp programming language 50 years ago. This hierarchical model is very simple and allows a simple annotation of the data. As in Lisp, the same tree notation used for data representation is also employed to write programs to transform tree structures into other tree structures. On top of this identity of data and program representation, in XML, the tree notation is also used to denote type information to validate XML data.

As is shown at the top of Figure 1.1, an arbitrary name between `<` and `>` symbols is given to a node of a tree. This is called a *start-tag*. Everything up until a corresponding *end-tag* (the same tag except that it starts with `</`) forms the content of the node, which can itself be a tree. Such trees (e.g. `wine`, `properties` and `color` in Figure 1.1) are called *elements*. Elements can also contain character data and even mix character data and elements (e.g. `food-pairing`). An XML element with no content can be indicated with an end-tag immediately following a start-tag and can be abridged as an *empty-element tag*: a start-tag with a terminating `/` (see `rating` in Figure 1.1). It is possible to have more than one element of the same name within the children of an element (see `tasting-note` in Figure 1.1). The ordering between children is important and is referred as the *document order*. Comments can be added to an XML file by means of a special element that starts with `<!--` and ends with `-->` (see end of the second line of Figure 1.1).

Additional information can be added to an element tag with *attribute* pairs comprising the name of the attribute (e.g. `format`), an equal sign and the corresponding character string value within double or single quotes (e.g. `"11"` or `'11'`). Attributes can also be added to an empty element (e.g. `rating`). Attribute names within a single start-tag must be unique and there is no ordering between the different attributes. Attributes within a single element are thus considered as a table or a dictionary in which the names are the keys and the strings, the corresponding values.

As illustrated in the middle part of Figure 1.1, these notations are equivalent to a tree data structure where each node is labelled with its name and attributes. Character data appears as leaf nodes. An empty element is a node with no sub-tree.

XML has the (well deserved) reputation of being verbose but it must be kept in mind that this notation is primarily aimed at communication between machines for which verbosity is not a problem but uniformity of notation is a real asset. In fact, humans should not be really required to type all these start-tags and end-tags. Indeed, many useful structural XML editors are now available which hide the verbosity, keeping only the important structural information or by displaying embedded tables instead of tags. Figure 1.2 shows alternative views of an XML file.

As has been shown by Lisp over the years, this tree notation is very general and can be used not only to represent data but also its processing. Programs for transforming XML tree structures into other tree structures can be written in XSL (eXtensible Stylesheet Language) stylesheets, which are a declarative notation for XML transformations also written in XML.

An important feature of XML, and one that differs from Lisp, is the a priori type checking that can be performed on the file and the validation that can be performed before processing. XML type information can be provided either through a DTD or a schema which offers a quite powerful and flexible type system. A schema is also written as an XML file which can itself be type checked. An alternative schema notation called RELAX NG will also be presented later in this document.

XML originated from the need for a flexible way of organizing natural language texts and thus its designers used standard representations of the characters—most often Unicode—and standard encodings such as UTF-8 or UTF-16, which will not be discussed here.

XML is also widely used in computing systems to systematize structured data as an alternative to databases. Many relational databases also offer XML specific features for indexing and searching. Because of the portability of its encoding and the fact that XML parsers are freely available, it is also used for many tasks requiring flexible data manipulation to transfer data between systems, as configuration files of programs and for keeping information about other files. This document will not present these applications but will

Figure 1.2. Web browser, summary and grid editor views of an XML file

At the top, the file of Figure 1.1 as displayed in a web browser; the right pointing triangle at the left of `<properties>` and the ellipsis between start and end tags indicate that this element is hidden by collapsing. By clicking on it, the right pointing triangle becomes a down pointing triangle and the tree is displayed in full. The bottom of the figure show alternative views of the same file available on a commercial XML editor (`<Oxygen/>`) in order to *hide* the tags from the user: on the left, a summary view similar to the one given in a browser but without the tags and on the right a grid editor view

```

▼ <wine name="M" code="00518712" format="11">
  ▶ <properties>...</properties>
  ▼ <origin>
    <country>Italy</country>
    <region>Abruzzo</region>
    <producer>Cantina Miglianico SCARL</producer>
  </origin>
  <rating stars="2"/>
  ▼ <food-pairing>
    Cold cuts,
    <bold>Meatloaf,</bold>
    Pizza
  </food-pairing>
  <food-pairing>Asian spicy pork burger</food-pairing>
  <price>9.95</price>
  <year>2004</year>
</wine>

```

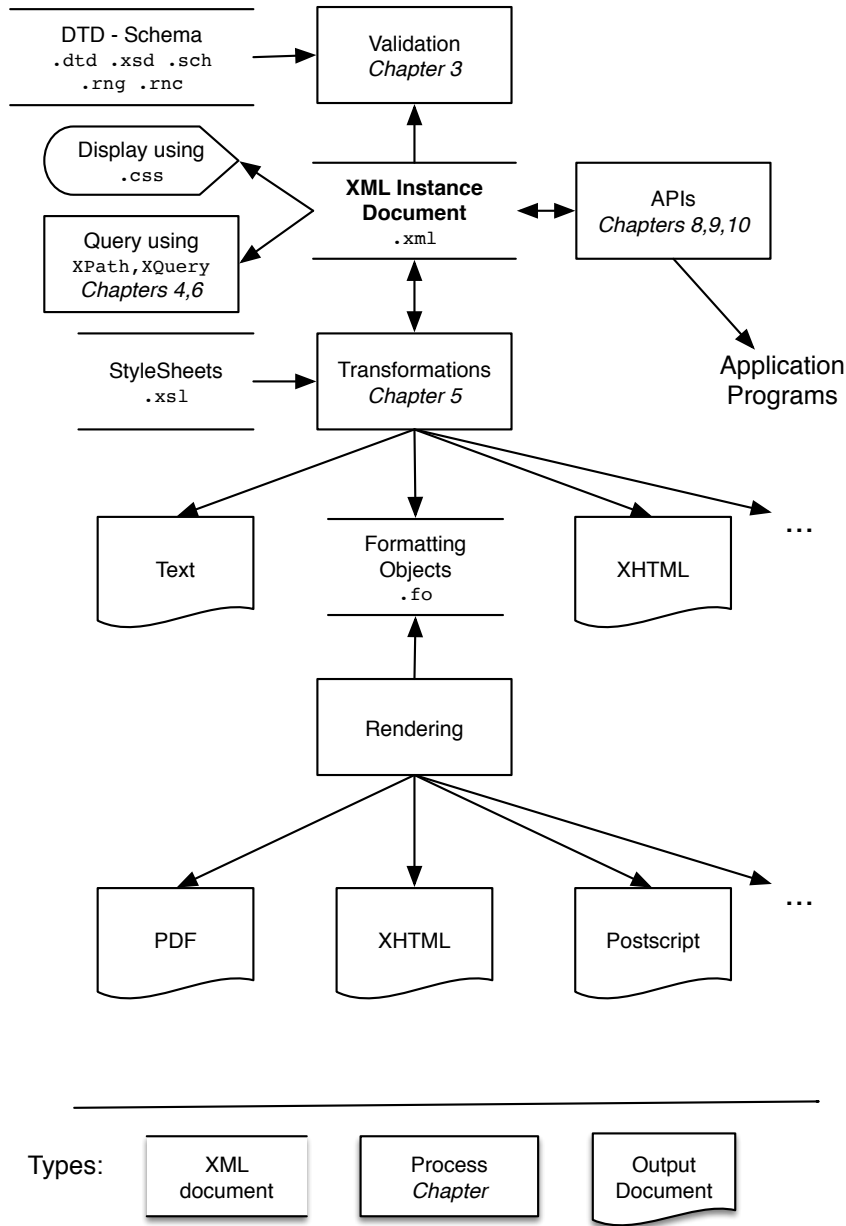
- wine "M" value of the attribute code is the SAQ.com code
- !- value of the attribute code is the SAQ.com code
- ▼ ● properties red
 - color red
 - alcoholic-strength 12
- ▼ ● origin Italy
 - country Italy
 - region Abruzzo
 - producer Cantina Miglianico SCARL
- rating "2"
- ▼ ● food-pairing Cold cuts,
 - bold Meatloaf,
 - food-pairing Asian spicy pork burger
 - price 9.95
 - year 2004

<?xml version="1.0" encoding="UTF-8"			
wine	@name	M	
	@code	00518712	
	@format	11	
	properties	color	red
		alcoholic-st...	12
	origin	country	Italy
		region	Abruzzo
		producer	Cantina Miglianico SCARL
	rating	@stars	2
	food-pairing (2 rows)	#text	bold
		#text	#text
	1	Cold cuts,	Meatloaf,
	2	Asian spicy pork burger	Pizza
	price	9.95	
	year	2004	

focus on a single one (creating a compact representation of an XML file) that will be used throughout so that one can appreciate the similarities and differences between the XML technologies presented.

Figure 1.3 presents the XML technologies we will describe in this report and their relations. The focus of the whole process is an *XML instance document* (towards the top of the figure) that contains the data. This XML document can be validated against a specification described either as a Document Type Description (DTD) or a XML Schema, itself another XML file. The validation process will be described in Chapter 3. Once validated, XML data can be used by application programs through specific Application Programming Interfaces (APIs) described in Chapter 8 and Chapter 9. XML data can also be processed by transformations (Chapter 5) written as stylesheets, a special kind of validated XML file, to create new XML, HTML, PDF or text files.

Figure 1.3. Relations between some XML technologies



For example, the XML file at the top of Figure 1.1 can be transformed with a stylesheet into an HTML one. The top of Figure 1.4 shows such a possible HTML output, displayed in a web browser shown at the bottom of the figure. This is the kind of tree-to-tree transformation for which XSLT was specifically designed. To better illustrate the power of the more general transformations that XSLT allows, we will show how to obtain a more *compact* form; it is shown in Figure 1.5 either as a text file (top) or in PDF (bottom) through a transformation using *Formatting Objects*. This compact notation is similar to that used in the Formal Description of XML [13]; it must be viewed as a programming exercise and not as a compression technique for XML files.

Figure 1.4. HTML source of the compact form and its rendering in a browser.

Representation of the tree in Figure 1.1 in source HTML and as it appears in a browser window. This HTML output (slightly reformatted here to fit in the page) was produced by our example stylesheet `compactHTML.xsl` shown in Example 5.7.

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head><title>HTML compaction of "Wine.xml"</title></head>
  <body>
    <ul><li><b>wine</b> name="M" code="00518712" format="11"
      <ul>
        <li><b>properties</b><ul>
          <li><b>color</b> red</li>
          <li><b>alcoholic-strength</b> 12</li>
        </ul>
        </li>
        <li><b>origin</b><ul>
          <li><b>country</b> Italy</li>
          <li><b>region</b> Abruzzo</li>
          <li><b>producer</b> Cantina Miglianico SCARL</li>
        </ul>
        </li>
        <li><b>rating</b> stars="2" </li>
        <li><b>food-pairing</b><ul>
          <li>Cold cuts, </li>
          <li><b>bold</b> Meatloaf,</li>
          <li>Pizza</li>
        </ul>
        </li>
        <li><b>food-pairing</b> Asian spicy pork burger</li>
        <li><b>price</b> 9.95</li>
        <li><b>year</b> 2004</li>
      </ul>
    </li></ul>
  </body>
</html>
```

- **wine** name="M" code="00518712" format="11"
 - **properties**
 - **color** red
 - **alcoholic-strength** 12
 - **origin**
 - **country** Italy
 - **region** Abruzzo
 - **producer** Cantina Miglianico SCARL
 - **rating** stars="2"
 - **food-pairing**
 - Cold cuts,
 - **bold** Meatloaf,
 - Pizza
 - **food-pairing** Asian spicy pork burger
 - **price** 9.95
 - **year** 2004

Figure 1.5. Compact forms in text and in PDF

Compact form of the tree in Figure 1.1 in text and PDF formats. These outputs were produced by the stylesheets of Example 5.9 and Example 5.10.

```
wine[@name[M]
  @code[00518712]
  @format[1l]
  properties[color[red]
    alcoholic-strength[12]]
  origin[country[Italy]
    region[Abruzzo]
    producer[Cantina Miglianico SCARL]]
  rating[@stars[2]]
  food-pairing[Cold cuts,
    bold[Meatloaf,]
    Pizza]
  food-pairing[Asian spicy pork burger]
  price[9.95]
  year[2004]]
```

wine	@name	M	
	@code	00518712	
	@format	1l	
	properties	color	red
		alcoholic-strength	12
	origin	country	Italy
		region	Abruzzo
		producer	Cantina Miglianico SCARL
	rating	@stars	2
	food-pairing		Cold cuts,
		bold	Meatloaf, Pizza
	food-pairing		Asian spicy pork burger
	price		9.95
year		2004	

This document is also an example of the use of these technologies because its source file is itself an XML document validated by a DocBook RELAX NG Schema and processed by two stylesheets: one produces a set of linked HTML files and another produces an XSL-FO file which is then rendered as a PDF file. This process is further described in Appendix C

This chapter has shown that XML is a flexible notation for adding information to *natural language* text but it is increasingly used in other areas as well. The raw XML is verbose and not very user-friendly but it can be hidden by appropriate tools. Programmers can also rely on freely available XML parsers and validators in order to get a well-organized data structure from an XML file.

This document tries to give an overall impression of some XML techniques and should not be considered a definitive or exhaustive manual. We will describe the main principles and present general rules and the intuition behind some of the technologies. For the sake of simplicity, we will sometimes be making *white lies* that seasoned XML experts could point out.

[T]he right abstraction [for XML ...] is a labeled tree of elements. Each element has an ordered list of children in which each child is a Unicode string or an element. An element is labeled with a two-part name consisting of a URI and local part. Each element also has an unordered collection of attributes where each attribute has a two-part name, distinct from the name of the other attributes in the collection, and a value, which is a Unicode string. That is the complete abstraction. [...] If you understand this, then you understand XML.

— James Clark, in [59], pages ix-x.

Chapter 2. Instance Document

Because there are many types of XML documents, either for transforming or validating data, an XML file that contains *data* is called an *instance document*. Any XML document must be *well-formed* which means that:

- all element start-tags and end-tags must be properly nested
- there should only be one top-level element in the file.

But there are also other peculiarities we will describe shortly in this chapter.

In the rest of this document, we will be using as input the XML instance files shown in Example 2.2 and Example 2.3. Their global organization is given in Example 2.1. They describe a wine cellar containing wine bottles defined in a separate wine catalog. This application was inspired by the *Livre de cave* example used by Benoît Habert in his book on the Common Lisp Object System (CLOS) programming [23]. Some of the lines in the examples are marked with numbered *callouts* (such as ❶) linked to an explanation at the end of each example.

The structure of our instance document files is the following:

- CellarBook.xml (Example 2.2) describes the cellar in four parts:
 - wine-catalog (line 9-❶) described in an external file Wine-Catalog.xml
 - owner (line 11-❷) name and address
 - location (line 21-❸) address of the cellar (if different from that of the owner)
 - cellar (line 27-❹) list of wine bottle lots (using codes from the wine catalog) and, for each, the quantity currently held in the cellar and the purchase date of the lot
- WineCatalog.xml (Example 2.3) gives the description of each wine product with a code that will correspond to codes of the cellar.

Example 2.1. Outline of CellarBook.xml including WineCatalog.xml in a different namespace

```
1 <cellar-book ...
    xsi:noNamespaceSchemaLocation="CellarBook.xsd"
    xmlns:cat="http://www.iro.umontreal.ca/lapalme/wine-catalog">
    <xi:include href="WineCatalog.xml" ... />
5
    <wine-catalog schemaLocation="http://www.iro.umontreal.ca/lapalme/wine-catalog Wine
        <wine name="Domaine de l'Île Margaux" ... >...</wine>
        <wine name="Riesling Hugel" ... >...</wine>
        <wine name="Château Montguéret" ... >...</wine>
10    <wine name="Mumm Cordon Rouge" ... >...</wine>
        <wine name="Prado Rey Roble" ... >...</wine>
    </wine-catalog>

    <owner>...</owner>
15    <location>...</location>
    <cellar>
        <wine code="C00043125">...</wine>
```

```

    <wine code="C00312363">...</wine>
    <wine code="C00871996">...</wine>
20    <wine code="C00929026">...</wine>
    </cellar>
  </cellar-book>

```

- ❶ Inclusion of the file `WineCatalog.xml`. The XML processor *replaces* this line at run-time by the content of the `wine-catalog` element shown here in **bold**.

Example 2.2. Excerpt of `CellarBook.xml`, the XML instance document holding the content of the cellar (Example D.1 shows the whole XML content)

```

1  <?xml version="1.0" encoding="UTF-8"?>                                ❶
  <!DOCTYPE cellar-book [<!ENTITY guy "Guy Lapalme" >                    ❷
    <!ENTITY eacute "&#xe9;" >
    <!ENTITY mtl "Montr&eacute;al" >
5    <!ENTITY GL "&guy;, &mtl;" >]>                                    ❸
  <cellar-book xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"    ❹
    xmlns:cat="http://www.iro.umontreal.ca/lapalme/wine-catalog"
    xsi:noNamespaceSchemaLocation="CellarBook.xsd">                    ❺
    <xi:include href="WineCatalog.xml"                                    ❻
10    xmlns:xi="http://www.w3.org/2001/XInclude"/>
    <owner>                                                                ❼
      <name>                                                                ❽
        <first>Jude</first>
        <family>Raisin</family>
15      </name>
        <street>1234 rue des Châteaux</street>
        <city>St-George</city>
        <province>ON</province>
        <postal-code>M7W 7S0</postal-code>
20      </owner>
      <location>                                                            ❾
        <street>4587 des Futailles</street>
        <city>Vallée des crus</city>
        <province>QC</province>
25      <postal-code>H3C 4J8</postal-code>
      </location>                                                         ❿
    <cellar>                                                                ❶❶
      <wine code="C00043125">
30        <purchaseDate>2005-06-20</purchaseDate>
        <quantity>2</quantity>
        <comment>                                                            ❶❷
          <cat:bold>&GL;</cat:bold>: should reorder soon
        </comment>
35      </wine>
      ...                                                                    ❶❸
      <wine code="C00929026">
        <purchaseDate>2003-10-15</purchaseDate>
        <quantity>1</quantity>
40      <comment>for <cat:bold>big</cat:bold> parties</comment>
      </wine>

```

```
</cellar>
</cellar-book>
```

- ❶ A *processing instruction* indicating the XML version used. Although XML version 1.1 exists, very few processors handle it, so most of the time `version="1.0"` is used. The encoding for the file, here it is UTF-8.
- ❷ Element `!DOCTYPE`, not a well-formed XML element, defines *entities* that can be used in the XML instance document. This notation will be explained further in Section 3.1 but for the moment they can be considered as *text macros* that will allow string substitutions before the XML file is processed. Substitution occurs when an entity is referred to by enclosing its name between `&` and `;`. For example, entity `guy` is replaced by `Guy Lapalme` when `&guy;` is encountered in the file.
- ❸ Entities can refer to other entities: `&GL;` will be replaced by `Guy Lapalme, Montréal`. When an entity declaration is followed by `SYSTEM` and the name of a file, then a reference to this entity is replaced by the content of the file.
- ❹ Starting tag indicating that this is an instance file to be validated with a given schema.
- ❺ Name of the file containing the schema for this instance file.
- ❻ Inclusion of another XML file describing the wine catalog.
- ❼ Description of the owner of the cellar book.
- ❽ Name of the owner of the cellar book.
- ❾ Location of the owner of the cellar book.
- ❿ Description of the cellar.
- ⓫ Information about a given wine, such as quantity and comments; the wine is identified by a code that must match one in the wine catalog.
- ⓬ As the `bold` element is defined in the schema associated with the catalog, it must be given the namespace prefix of the catalog.
- ⓭ Description of another wine of the cellar.

Example 2.3 is the catalog of available types of wines storing information such as their properties (color, alcoholic strength), their origin, their price and their year of production. This information was inspired by data found on the web site of the Société des Alcools du Québec. Other information such as the name, the code and format are given as attributes within the start-tag. While the value of an element can be an arbitrarily complex tree of elements, attribute values can only be single string values. Strings for attribute values must be delimited by either matching `'` or `"`. These delimiters have the same meaning and this convention is convenient when embedding a quote of one type within a string value. In case the two types of quotes are needed within a single string, one can use the predefined entities `'` and `"` (explained in Section 3.1).

The structure of an XML instance file may seem arbitrary and, in a sense, it is. In order to make sure that its processing is efficient, it is important that the structure of the information be in the right format (i.e. embedded within the correct tags and in the correct order) and that all the mandatory information be present. This verification could be performed by the program using the information but it would more interesting to detect errors or lack of information when the instance file is created. Thus the program needing the data can be sure that the file structure follows the expected format. This validation process, similar to the static type checking for a programming language, is explained in the next chapter but before, we will look at namespaces, another important concept in XML instance documents.

Example 2.3. Excerpt of WineCatalog.xml, the XML instance document holding the content of the wine catalog, it is included in Example 2.2 (Example D.2 shows the XML content)

```
1 <wine-catalog xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ❶
  xsi:schemaLocation="http://www.iro.umontreal.ca/lapalme/wine-catalog ❷
    WineCatalog.xsd"
  xmlns="http://www.iro.umontreal.ca/lapalme/wine-catalog"> ❸
5 <wine name="Domaine de l'Île Margaux" ... > ❹
  <properties>
    <color>red</color>
    <alcoholic-strength>12.5</alcoholic-strength>
    <nature>still</nature>
10 </properties>
  <origin>
    <country>France</country>
    <region>Bordeaux</region>
    <producer>
15      SCEA Domaine de L'Île Margaux (B.P. 5)
    </producer>
  </origin>
  <comment>Ready for drinking now</comment>
  <food-pairing>
20    Accompanies <emph>Bordelaise ribsteak</emph>, ❺
      <bold>pork with prunes</bold> or magret de canard.
  </food-pairing>
  <price>22.80</price>
  <year>2002</year>
25 </wine>
  ...
  <wine name="Prado Rey Roble" ... >
    <properties>
      <color>red</color>
30      <alcoholic-strength>12.5</alcoholic-strength>
      <nature>still</nature>
    </properties>
    <origin>
      <country>Spain</country>
35      <region>Old Castille</region>
      <producer>Real Sitio de Ventosilla SA</producer>
    </origin>
    <price>35.25</price>
    <year>2002</year>
40 </wine>
</wine-catalog>
```

- ❶ Start of the wine catalog description.
- ❷ Namespace and file name of the Schema for this instance file.
- ❸ Namespace for the wine catalog, useful to differentiate the wine elements that appear both in the catalog and the cellar book.
- ❹ Description of a wine.
- ❺ Some words can be emphasized by surrounding them with bold and emph tags.

2.1. Namespaces

Until now we wrote XML element name as a simple identifier, but in fact the *full* name of an element (or an attribute) is an identifier combined with a namespace which is an arbitrary string. If the string is empty, then we consider that the element is in the empty namespace. But usually a namespace is a long string that we would like to be unique for an individual or a corporation. One convenient way of ensuring unique name is to use strings that are similar to urls; the reasons for this will be explained in Chapter 7, but for the moment we will use short strings to simplify the explanations.

The namespace for an element is specified using the `xmlns` as illustrated in the first tree in Example 2.4 (line 7-❶). Elements named `b`, `c` and `e` are in the `mary` namespace, while `a` and `d` are in the `john` namespace.

When the `xmlns` attribute is not specified for an element, it is looked up in the ancestors of the node (first the parent, then parent of the parent and so on). The first value associated with an `xmlns` attribute encountered is assigned to the current element. If no `xmlns` attribute is encountered then the current element is considered to be in the empty namespace (equivalent to the declaration of `xmlns=""`). The second tree (line 18-❷) shows how the namespaces for elements `tree`, `c` and `d` are inherited by this rule to give equivalent element names as the ones in the first tree.

But using `xmlns` can become burdensome and error-prone, so the usual way of specifying namespaces is by declaring a prefix, usually only a few letters long, for a given namespace and then using the prefix in front of the element names. The prefix is specified with `xmlns:prefix=string` which defines the prefix as specifying the namespace for the entire subtree of the element, including itself. In the third tree (line 30-❸), in element `a`, `m` is declared as a prefix for denoting the `mary` namespace, which is then used for elements `b`, `c` and `e`. The prefix string has no significance in itself outside of referencing the full string of the namespace. Different prefixes could in principle be used in different parts of the same XML file for referencing the same namespace although this would be less convenient for a human reader. When most of the elements of an XML tree are to be put in a given namespace, it is often advantageous to declare the namespace as the default namespace in the root element tag, so that all elements in the tree without a prefix will be automatically put in the given namespace.

Example 2.4. [NamespaceExample.xml] A simplistic example of declaration and use of namespaces

Three equal XML trees showing different ways of assigning namespaces to elements. The convention used in the third tree is most often used.

```

1 <?xml version="1.0" encoding="UTF-8"?>
  <!-- simple illustration of the definition of namespaces:
     all three tree elements are equal -->
  <trees-with-namespaces>
5     <!-- Names of namespaces are short here for simplicity,
       but they should be unique long strings... -->
       <tree no="1" xmlns="">
         <a xmlns="john">
           <b xmlns="mary">
10              <c xmlns="mary">ghi</c>
           </b>
           <d xmlns="john">
```

❶


```
        <e xmlns="mary">jkl</e>
    </d>
15    </a>
    </tree>
    <!-- xmlns declaration are inherited from the ancestors -->
    <tree no="2">
        <a xmlns="john">
20            <b xmlns="mary">
                <c>ghi</c>
            </b>
            <d>
                <e xmlns="mary">jkl</e>
25            </d>
        </a>
    </tree>
    <!-- prefix declarations span the tree and simplify notation -->
    <tree no="3">
30        <a xmlns="john" xmlns:m="mary">
            <m:b>
                <m:c>ghi</m:c>
            </m:b>
            <d>
35            <m:e>jkl</m:e>
            </d>
        </a>
    </tree>
</trees-with-namespaces>
```

- ❶ Long hand definition of namespaces in each element
- ❷ Use of inheritance for the `xmlns` attribute.
- ❸ Definition of a prefix for the elements that are not in the same namespace as the one of the parent.

Namespaces allow a graceful combination of independent XML files. Coming back to our running example (Example 2.1), let us see how these principles are used in a larger and less artificial context: Example 2.2 includes Example 2.3 via the element `xi:include`. These files both use the `wine` element, but in different ways: in Example 2.3, `wine` refers to a description of a wine type while in Example 2.2 `wine` refers to a batch of bottles. Both references must be distinguished from one another in order to validate them with the appropriate XML Schema. In such a simple case, it would be an easy matter, and probably a better design, to have different names for these two concepts but we want to illustrate the use of namespaces in a small scale example. A similar name clash could happen if we wanted to combine independently created XML files.

For example, in the root tag of line 6-❹ of Example 2.2, two namespace prefixes are defined: `xsi` and `cat`, for which are given two *arbitrary* unique identifiers that will be used to distinguish their namespaces. Most often, identifiers of namespaces are URLs (URIs more precisely) because the authors of an XML file use a URL linking to a web site they own. If authors take care not to use the same URL for different purposes, this pretty much guarantees the uniqueness of the namespaces. This *does not* necessarily means that the URLs used as names for namespaces do exist. It must be stressed that the URL notation is nothing more than a useful convention, although this name can also be used by validators as a *hint* to find the corresponding schema.

By default, names without prefixes are defined in the *empty* namespace or in the value assigned to the `xmlns` attribute. To create elements in a specific namespace, we assign a *default namespace* like we did at the start of Example 2.3 by specifying a value for the `xmlns` attribute (line 4). In principle, any element can set a value for the `xmlns` attribute to change the default namespace or to set the prefix of new namespaces for nested elements. As namespace prefixes are inherited, the search for the URI corresponding to a prefix starts from the current element and follows the parent links in the tree until it finds a corresponding prefix declared as a value of a `xmlns` attribute.

As shown in Example 2.1, the declaration of namespaces is most often done at the root element of the file. In this listing, elements in italics are in a different namespace. A non-italicized element must use a prefix to refer to an element in italics. Within italicized code, no prefix is necessary because the namespace has been given a null prefix (line 4) within the box. It is possible to define a namespace for any element (which will also apply to its subelements) but this makes it hard for the human reader to be aware of the current namespace of an element. However, a *namespace aware* XML processor has no problem because a namespace is associated with each element. For example, in Example 2.2, `cat:bold` designates the `bold` element in the `http://www.iro.umontreal.ca/lapalme/wine-catalog` namespace. All elements in the Example 2.3 also have the same namespace; so `bold` elements (line 20-) are the *same*: i.e. when, as will be explained later, they will be processed by an XML system, they will be identified as being of the same type.

The use of namespaces will be better understood once we have seen their use in validation with schemas (Section 3.2.3). An excellent short introduction to the concept of namespace can be found in [59], pp. 160—166. Namespaces are fundamental in the context of the semantic web as discussed in Chapter 7.

Chapter 3. Document Validation

As we have mentioned in the previous section, an XML file must be *well-formed* in order to be processed correctly. XML designers have created a thorough checking method called *validation* that verifies whether elements of an XML file are well-formed and, furthermore, ensures that their ordering and nesting obey certain rules. These rules are specified by a DTD or an XML Schema. This validation is performed prior to any further processing so that programs that process an XML file do not waste time checking for such errors. An application is even allowed to stop any processing if it encounters an invalid XML file.

The author of an XML file can usually be warned of the invalidity of his XML file at creation time. This validation can be done either within the XML text editor itself (e.g. XMLSpy [3], <oxygen/> [52] or the nXML mode in Emacs [19]) or by an external validator program (e.g. Xerces [4] or XSV [46]). XML editors can also play an active role in the creation of valid XML files, by suggesting at each point valid completions (acceptable elements, attributes or values) depending on the DTD or the XML Schema.

XML, like its ancestor SGML, defines the validation of a file with respect to a *Document Type Declaration* (DTD) declared at the start of the file. Most often, the DTD is an accompanying external document that allows different files to follow the same rules by sharing it. A DTD is relatively simple to define but the validation rules it can enforce are quite rudimentary because they can only define constraints on the nesting of elements and perform simple checking on values of attributes.

In order to validate the content of elements, XML designers have defined a more elaborate type system called a *Schema* which can be used in at least two technologies: XML Schema presented in Section 3.2 and RELAX NG described in Section 3.3. On top of these grammar-based validation techniques, we will also present Schematron, a different approach to validation by means of rules that are evaluated on the instance document.

3.1. Document Type Definition (DTD)

A DTD is a notation to define elements that are allowed to appear in an XML file as well as the type of information they can contain. Table 3.1 gives an overview of some of the more frequent definitions of elements, attributes and entity that can be made in a DTD. These definitions are *pseudo-XML* tags in the sense that they look like XML start-tags without their corresponding end-tags. For mainly historical reasons, DTDs are not well-formed XML files. The types for DTDs are most often given as either:

- (#PCDATA) (*Parsed Character DATA*) which corresponds to character string information; *parsed* means that the character data can contain entity references as explained below;
- a regular expression in parentheses involving other element names.
- an element can also be a *Character Data fragment* enclosed between `<![CDATA[and]]>` which inserts its content verbatim without any interpretation. This is often used to display XML content without having to escape the `<` and `&`.
- the value of an attribute can only be a string (CDATA) but a DTD allows for some special case that enable some simple validations:
 - ID : a string starting with a letter or an underline (`_`) followed by a sequence of letters, numbers, underlines, periods (`.`) and dashes (`-`). All IDs must be unique in the document.
 - IDREF : a string that must refer to an existing ID in the document.

- a list of strings enclosed in parenthesis and separated by a | . Note that this validation can only be applied on an attribute and not on the value of an XML element."

Table 3.1. DTD syntax reminder

<code><!DOCTYPE rootElement SYSTEM «"»file name«"» «>» {[!ENTITY *]}? «>»</code>
<code><!ELEMENT NCName «(» {#PCDATA « » }? regexpOf element name «)» «>»</code>
<code><!ELEMENT NCName (#PCDATA) «>»</code>
<code><!ELEMENT NCName EMPTY «>»</code>
<code><!ATTLIST elementNCName attributeNCName declValue default «>» declValue = CDATA ID IDREF «(» CNAME+ «)» default = {#REQUIRED #IMPLIED}</code>
<code><![CDATA[...]]«>»</code>
<code><!ENTITY name «"» ... «"» «>»</code>
<code><!ENTITY % name «"» ... «"» «>»</code>
<code><!ENTITY name SYSTEM «"»file name«"» «>»</code>

A reminder of the subset of the DTD syntax used in Example 3.1 and Example 3.2. CDATA is character data as is, but PCDATA is *parsed* character data that can contain references to entities. Names in italics refer to other elements. declValue and default are not part of the DTD syntax, they are only handy abbreviations in this table. Regular expressions syntax is given in Table B.1. Terminal identifiers are indicated in bold and terminal symbols are enclosed in «chevrons».

Example 3.1 is a validating DTD for the XML instance document given in Example 2.2. Elements are defined with an !ELEMENT tag, see wine (line 5-② of Example 3.1), containing a regular expression indicating constraints on its children elements: it is an element with up to four children elements in sequence: purchaseDate, quantity, rating and comment, the last two being optional. purchaseDate (line 6-③) can contain character data and no other elements. A cellar (line 3-④) is a list (possibly empty) of wine elements. A name (line 12-⑤) is a non-empty list of either a first, initial or family in any order; as the order of these elements is not fixed within a name element, we must allow an infinite repetition of these without guaranteeing that the same element is not given . This illustrates some of the limitations on the types of constraints that can be easily represented with a DTD. The alternative would be to give explicitly all permutations of the three elements within a name.

Attributes are defined using !ATTLIST tags indicating the element to which they belong, their name, their type and whether they are mandatory (#REQUIRED) or optional (#IMPLIED). See for example the !ATTLIST for the code attribute of the wine element (line 10-⑥).

A DTD can also contain definitions of entities that act as text macros that are replaced textually either in the instance document or in the DTD itself. Entities whose definitions start with <!ENTITY such as guy (already illustrated in Example 2.2) define textual replacements where they are *called*, i.e. where they appear between & and ; . There are five predefined entities :

<code>&lt;</code> ;	a <i>less-than sign</i> (<) in an XML file because < is reserved to indicate the start of a tag;
<code>&amp;</code> ;	As we have introduced the ampersand to indicate the start of an entity, we need a way to insert an ampersand (&);
<code>&quot;</code> ;	for inserting a " ;
<code>&apos;</code> ;	for inserting a ' ;

> by symmetry with < even though it is not strictly needed.

Entities are replaced in an XML file when the file is first read in memory before its interpretation; that means that those entities are replaced within any context even within strings.

Entity replacements can also be used within the DTDs themselves to modularize them. These types of entities are called *parameter entities* and they are distinguished from *ordinary* entities by adding a percent sign preceding the *name* of the parameter entity and its definition, see `address` (see line 26-⑨). Its call is preceded by a percent sign instead of an ampersand see `owner` (line 32-⑩) and `location` (line 33-⑩). Another special kind of entity, introduced by SYSTEM, refers to a file such as `wine-catalog` (line 36-⑩). This entity can then be used to include a file as is shown on the last line of the `cellar` DTD Example 3.1 which includes the `wine catalog` DTD Example 3.2. As the parameter and system entities are replaced when the DTD is built, they cannot be used in instance documents so no entity is needed to enter a percent.

Example 3.1. [CellarBook.dtd] DTD for the cellar book, validation of the instance file in Example 2.2

ELEMENTs and ATTLISTs are independent, indentation is ignored by the DTD processor, it is used here for the human reader only, to highlight some inclusion dependencies.

```

1  <?xml version="1.0" encoding="UTF-8"?>

    <!ELEMENT cellar (wine)* >                                     ①

5  <!ELEMENT wine (purchaseDate,quantity,rating?,comment?) >    ②
    <!ELEMENT purchaseDate (#PCDATA) >                          ③
    <!ELEMENT quantity (#PCDATA) >                               ④
    <!ELEMENT rating EMPTY >
        <!ATTLIST rating stars CDATA #IMPLIED>
10   <!ATTLIST wine code IDREF #REQUIRED >                       ⑤

    <!ELEMENT name (first | family | initial)+ >                 ⑥
        <!ELEMENT first (#PCDATA) >
        <!ELEMENT family (#PCDATA) >
15   <!ELEMENT initial (#PCDATA) >

    <!ELEMENT cellar-book (wine-catalog, owner, location, cellar) > ⑦

    <!-- [general] entities for use in instance document -->
20 <!ENTITY guy "Guy Lapalme" >                                  ⑧
    <!ENTITY eacute "é" >
    <!ENTITY mtl "Montr&eacute;al" >
    <!ENTITY GL "&guy;, &mtl;" >

25 <!-- parameter entities for use within a DTD -->
    <!ENTITY % address "(street, city, province, postal-code)" > ⑨
        <!ELEMENT street (#PCDATA) >
        <!ELEMENT city (#PCDATA) >
        <!ELEMENT province (#PCDATA) >
30   <!ELEMENT postal-code (#PCDATA) >

    <!ELEMENT owner (name,%address;) >                           ⑩

```

```
<!ELEMENT location %address; > ❶
```

```
35 <!-- system entity for including a file -->
<!ENTITY % wine-catalog SYSTEM "WineCatalog.dtd" > ❷
%wine-catalog;
```

- ❶ A cellar is a possibly empty list of wines.
- ❷ A wine is defined by a purchase date, a quantity, an optional rating and an optional comment.
- ❸ The purchase date is a string.
- ❹ A quantity is also a string.
- ❺ A code must refer to an id defined in the wine catalog.
- ❻ The name is a non-empty list consisting of a first name, a family name and an initial; this somewhat loose definition of a name illustrates some of the limitations of the the regular expression mechanism in the DTD.
- ❼ The cellar book is composed of the wine catalog, the owner, the location, and the cellar itself.
- ❽ Some definitions of entities and how they can be composed.
- ❾ Parameter entities can be used in the definition of other entities.
- ❿ Use of a parameterized entity defined on line 26-❹.
- ⓫ Another use of the same parameterized entity.
- ⓬ A parameterized entity definition for file inclusion; here it is used in the following line to include the DTD definitions in the wine catalog.

We now look at the validation of the wine catalog (Example 3.2). Given the fact that all element names must be unique in a DTD (there are no namespaces in DTDs), we must give a different name to the wine element of Example 3.1. Here we decided to call it `cat-wine` (line 4-❶). The attribute `format` (line 10-❷) shows an example of an enumeration of values from which the attribute value must necessarily be chosen. The link between the `wine` and the `cat-wine` elements is made using the `code` (line 13-❸ of Example 3.2) of type ID and its reference in line 10-❹ of Example 3.1 which is of type IDREF. In an XML file, all values of type ID must be distinct and values of type IDREF must refer to an existing ID.

Example 3.2. [wineCatalog.dtd] DTD to validate the wine catalog, included in Example 3.1

```
1 <?xml version="1.0" encoding="UTF-8"?>
  <!ELEMENT wine-catalog (cat-wine*) >

  <!ELEMENT cat-wine (properties, origin, ❶
5      (tasting-note?,food-pairing?,comment?)*,
      price,year) >
    <!ATTLIST cat-wine name CDATA #REQUIRED >
    <!ATTLIST cat-wine appellation CDATA #IMPLIED >
    <!ATTLIST cat-wine classification CDATA #IMPLIED >
10    <!ATTLIST cat-wine format (375ml | 750ml | 1l | magnum ❷
        | jeroaboam | rehoboam | mathusalem | salmanazar
        | balthazar | nabuchodonosor) #REQUIRED >
    <!ATTLIST cat-wine code ID #REQUIRED> ❸
    <!ELEMENT properties (color,alcoholic-strength,nature?) >
15    <!ELEMENT color (#PCDATA) >
```

```

        <!ELEMENT alcoholic-strength (#PCDATA) >
        <!ELEMENT nature (#PCDATA) >
    <!ELEMENT origin (country,region,producer) >
        <!ELEMENT country (#PCDATA) >
20      <!ELEMENT region (#PCDATA) >
        <!ELEMENT producer (#PCDATA) >

        <!ENTITY % Comment "(#PCDATA | emph | bold)*" >
        <!ELEMENT emph (#PCDATA) >
25      <!ELEMENT bold (#PCDATA) >

        <!ELEMENT comment %Comment; >
        <!ELEMENT tasting-note %Comment; >
        <!ELEMENT food-pairing %Comment; >
30

        <!ELEMENT price (#PCDATA) >
        <!ELEMENT year (#PCDATA) >

```

- ❶ A wine entry in the catalog is a list of properties, then the wine origins, followed by an optional list of tasting-notes, food-pairing suggestions and comment; finally the price and year have to be given.
- ❷ The format is given as a list of literal values from which the actual value of the attribute must be chosen.
- ❸ The unique code for the wine that must be matched by a reference at line 10-❹ of Example 3.1

3.1.1. Associating an Instance File with a DTD

The link between a DTD and an XML file that it validates can be done externally using an XML Editor or by a command line argument to a validation program. But most DTD validators rely on a `!DOCTYPE` element at the start of the XML file. For example, one can use declarations such as the following:

```

1      <!DOCTYPE cellar-book SYSTEM "CellarBook.dtd">
      <cellar-book>
          <wine-catalog>
2          ...
          </wine-catalog>
          ...
      </cellar-book>

```

The root element of the XML instance document is given as the second value, `SYSTEM` in third and a reference to the DTD file in fourth.

3.2. Schema

As we have seen in the previous section, a DTD defines constraints on the order and nesting of elements in an XML file, but the type of constraints is limited and does not allow validation of the character content of elements, except for enumeration values and `ID/IDREF`. There are also other drawbacks: all element names in a DTD must be unique and thus combining separately developed DTDs can become quite cumbersome. Moreover, the DTD file is not a well-formed XML file, thus one cannot easily use an XML tool to create or process it. This is why XML Schema has been introduced with a comprehensive set of elementary types and a way to combine them to create new types. The concept of namespaces (presented in Section 2.1) is also used in order to facilitate the combination of independent files without name clashes.

A Schema is a well-formed XML file (usually with a `.xsd` extension) that defines types used to validate the elements of an XML file. In a way similar to variable declarations in a programming language, we can define types. We follow the Java convention of starting type identifiers with an upper case letter. Element identifiers start with a lower case letter. In a name comprising more than one word, each word starts with an uppercase letter. No underscore or dash are used. For most elements, we define a type having the same name as the element, but capitalized, instead of using inline definitions of embedded elements. In a Schema, there are two kinds of types: simple and complex. Simple types define constraints on the text content of an element which cannot contain any element. A complex type can contain nested elements.

There are many different ways of organizing a Schema as described by Van der Vlist [60]. One can either use a *Russian doll* approach in which a single element is defined with its nested elements defined internally. Another approach is to use a *bottom-up* approach in which the elements are defined before they are used in more complex elements. It is also possible to use a *top-down approach* that first defines the higher level elements before defining the lower level elements. All these styles of definitions are possible and we will sometimes use a mix of them in order to show some features of XML Schema.

Table 3.2 presents the XML elements we use in our example to define the types needed for the validation of our wine catalog and cellar book. Since a schema is itself an XML file, it is important to distinguish the elements defining the Schema from the elements being defined. This is ensured by using a different namespace for the defining element (*definiens*) such as `xs:` as prefix (`xsd:` is also commonly used). A defined element (*definiendum*) does not have a prefix, it is in the default namespace. Contrarily to a DTD, an XML Schema is a valid XML file and it can be validated using the XML Schema of XML Schemas, usually provided with in all XML editors.

Table 3.2. XML Schema syntax reminder

<code><xs:schema targetNamespace="URI"> <i>xs:import</i>* {<i>xs:simpleType</i> <i>xs:complexType</i> <i>xs:element</i> <i>xs:group</i>}* </xs:schema></code>
<code><xs:import namespace="URI" schemaLocation="URI"/></code>
<code><xs:simpleType name="NCName"> <i>xs:restriction</i> </xs:simpleType></code>
<code><xs:complexType name="NCName" mixed="true"> {<i>xs:choice</i> <i>xs:sequence</i> <i>xs:group</i>}? <i>xs:attribute</i>* </xs:complexType></code>
<code><xs:element name="QName" type="TName"/></code>
<code><xs:element name="QName" ref="ENAME"/></code>
<code><xs:element name="QName"> {<i>xs:simpleType</i> <i>xs:complexType</i>}? {<i>xs:unique</i> <i>xs:key</i> <i>xs:keyref</i>}* </xs:element></code>
<code><xs:sequence {min max}occurs="nonNegativeInteger unbounded"> {<i>xs:element</i> <i>xs:choice</i> <i>xs:sequence</i> <i>xs:group</i>}* </xs:sequence></code>
<code><xs:choice {min max}occurs="nonNegativeInteger unbounded"> {<i>xs:element</i> <i>xs:choice</i> <i>xs:sequence</i> <i>xs:group</i>}* </xs:choice></code>
<code><xs:group name="NCName"> {<i>xs:choice</i> <i>xs:sequence</i>}* </xs:group></code>
<code><xs:attribute name="NCName" type="TName" use="required"/></code>
<code><xs:restriction base="TName"> <<i>xs</i>:{<i>max/min</i>}{<i>in/ex</i>}clusive value="anySimpleType"/> <<i>xs</i>:{<i>max/min</i>}length value="nonNegativeInteger" <pattern value="regExp" <enumeration value="anyValue" </xs:restriction></code>
<code><xs:{unique key} name="NCName"> <i>xs:selector</i> <i>xs:field</i>+ </xs:{unique key}></code>
<code><xs:keyref name="NCName" refer="NCName"> <i>xs:selector</i> <i>xs:field</i>+ </xs:keyref></code>
<code><xs:{selector field} xpath="XPathExpr"/></code>

A reminder of the subset of the XML Schema syntax used in Example 3.3 and Example 3.4. Names in italics refer to other elements. NCName (non-colonized name) is a name without a namespace prefix. Regular expressions syntax is given in Table B.1.

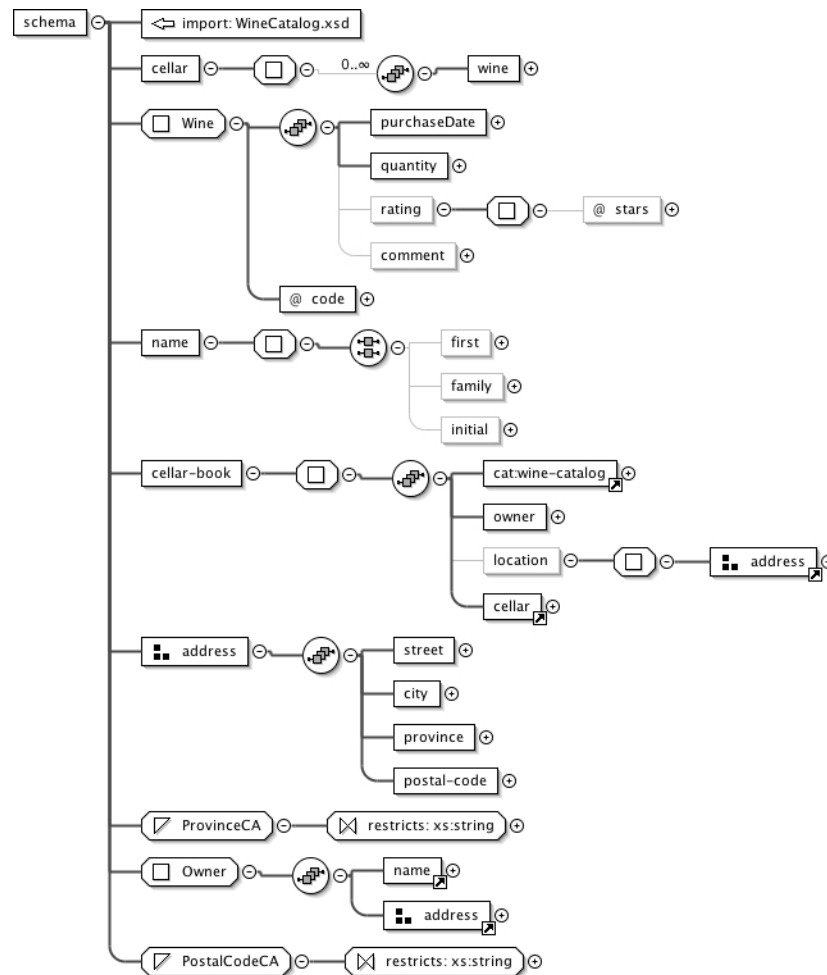
An XML Schema has an `xs:schema` element as root which can contain different kinds of definition elements.

- `xs:import` allows the combination of different schemas into a single one; in our case, we have a schema for the wine catalog which is imported into that of the cellar book.
- `xs:simpleType` defines additional constraints on predefined types; this is explained further in Section 3.2.1.
- `xs:complexType` defines a new type in terms of a choice or a sequence between other types.
- `xs:group` gives a name to an incomplete type that can be used as building block for other types.
- `xs:element` is the fundamental way of defining an element that can appear in an instance file. It can be given either with a name and a type. It can refer to another element definition or it can be defined with an anonymous simple or complex type followed by keys and keyrefs definitions.
- `xs:sequence` combines other elements by making sure that they occur sequentially.
- `xs:choice` combines other elements by making sure that only one of them occurs.
- `xs:all` combines other elements by making sure that they all occur but in any order.
- `xs:attribute` gives the name and the simple type associated with an attribute. Attributes are not ordered and optional unless their are given the value `required` to their use attributes. Note that `xs:attributes` are given at the end of the type definition, even though they appear in the start-tag.
- `xs:restriction` gives range, pattern constraints on the value of a simple type. `enumeration` is to limit the allowed value to one of a given list.
- `xs:key`, `xs:unique` and `xs:keyref` define cooccurrence constraints that generalize the notion of ID/IDREF. They will not be explained in this document.

In the rest of this section, we first give in Example 3.3 and Example 3.4 the XML Schemas that correspond respectively to the DTDs of Example 3.1 and Example 3.2. Figure 3.1 gives the overall structure of the XML Schema of Example 3.3. Figure 3.2 gives the overall structure of the Schema in Example 3.4. As we will see, the validation of the text content of the elements can be much more thorough with an XML Schema than with a DTD.

We will then explain the structure of the type system: first simple types (Section 3.2.1) and then complex types (Section 3.2.2).

Figure 3.1. Graphical view of the schema for the cellar book



A name in a rectangular box is an *element* name or, if preceded by @, an *attribute* name. A *complex type* name is preceded by a square and a *simple type* by a triangle. A *sequence* is shown with 4 dots horizontally aligned in an hexagon and a *choice* with 4 dots aligned vertically (see Figure 5.2 for an example). A + after a box indicates that further details have been omitted. Three small squares in front of an element name either indicates that its definition will be referred to somewhere else in the schema; the reference is indicated by a small arrow at the bottom right of the rectangle. This figure was produced by the <Oxygen/> XML editor from the XML Schema file given in Example 3.3.

Example 3.3. [CellarBook.xsd] XML Schema for the cellar book, validation of the instance file in Example 2.2.

This file can be compared with the DTD shown in Example 3.1.

```
1 <?xml version="1.0" encoding="UTF-8"?>
  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:cat="http://www.iro.umontreal.ca/lapalme/wine-catalog">

5    <xs:import namespace="http://www.iro.umontreal.ca/lapalme/wine-catalog"
      schemaLocation="WineCatalog.xsd"/>

    <xs:element name="cellar">
      <xs:complexType
10        <xs:sequence minOccurs="0" maxOccurs="unbounded">
          <xs:element name="wine" type="Wine"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>

15    <xs:complexType name="Wine">
      <xs:sequence
20        <xs:element name="purchaseDate" type="xs:date"/>
        <xs:element name="quantity" type="xs:nonNegativeInteger"/>
        <xs:element name="rating" minOccurs="0">
          <xs:complexType
25            <xs:attribute name="stars" type="xs:positiveInteger"/>
          </xs:complexType>
        </xs:element>
        <xs:element name="comment" type="cat:Comment" minOccurs="0"/>
      </xs:sequence>
      <xs:attribute name="code" type="xs:IDREF" use="required"/>
    </xs:complexType>

30    <xs:element name="name">
      <xs:complexType
35        <xs:all>
          <xs:element name="first" type="xs:string" minOccurs="100"/>
          <xs:element name="family" type="xs:string" minOccurs="0"/>
          <xs:element name="initial" type="xs:string" minOccurs="0"/>
        </xs:all>
      </xs:complexType>
    </xs:element>

40    <xs:element name="cellar-book">
      <xs:complexType
45        <xs:sequence>
          <xs:element ref="cat:wine-catalog"/>
          <xs:element name="owner" type="Owner"/>
          <xs:element name="location" minOccurs="0">
            <xs:complexType
50              <xs:group ref="address"/>
            </xs:complexType>
          </xs:element>
          <xs:element ref="cellar"/>
        </xs:sequence>
      </xs:complexType>
```

```

55     </xs:element>

        <xs:group name="address"> ❷
            <xs:sequence>
                <xs:element name="street" type="xs:string"/>
                60     <xs:element name="city" type="xs:string"/>
                <xs:element name="province" type="ProvinceCA"/>
                <xs:element name="postal-code" type="PostalCodeCA"/> ❸
            </xs:sequence>
        </xs:group>

65     <xs:simpleType name="ProvinceCA">
        <!-- http://www.canadapost.ca/tools/pg/manual/b03-e.asp#c012 -->
        <xs:restriction base="xs:string">
            <xs:enumeration value="AB"/>
            70     <xs:enumeration value="BC"/>
            <xs:enumeration value="MB"/>
            <xs:enumeration value="NB"/> ❹
            <xs:enumeration value="NL"/>
            <xs:enumeration value="NT"/>
            75     <xs:enumeration value="NS"/>
            <xs:enumeration value="NU"/>
            <xs:enumeration value="ON"/>
            <xs:enumeration value="QC"/>
            <xs:enumeration value="SK"/>
            80     <xs:enumeration value="YT"/>
        </xs:restriction>
    </xs:simpleType>

    <xs:complexType name="Owner"> ❺
        85     <xs:sequence>
            <xs:element ref="name"/>
            <xs:group ref="address"/>
        </xs:sequence>
    </xs:complexType>

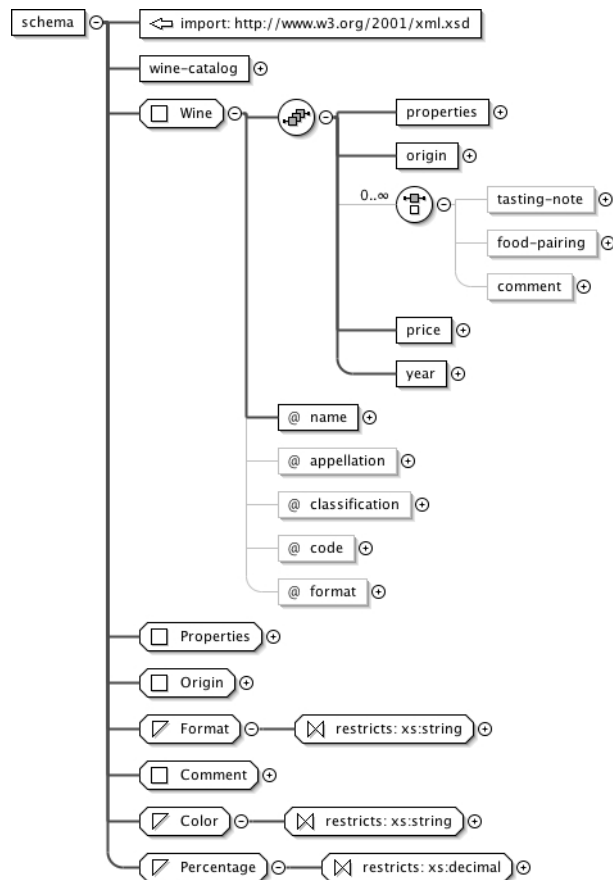
90     <xs:simpleType name="PostalCodeCA"> ❻
        <xs:restriction base="xs:string">
            <xs:pattern value="[A-Z][0-9][A-Z] [0-9][A-Z][0-9]"/>
        </xs:restriction>
95     </xs:simpleType>
</xs:schema>

```

- ❶ Gets the types defined in the schema for the wine catalog, which uses a different namespace, the one identified by the `cat` prefix.
- ❷ Element `cellar` is a possibly empty list of wine elements of type `Wine` defined at line line 16-❹.
- ❸ Definition of element `wine` of type `Wine`.
- ❹ The type `Wine` for the cellar-book defines a set of bottles; it is defined with their date of purchase, the number of bottles, an optional rating given as a number of stars and an optional comment whose type is defined in the catalog in a different namespace.
- ❺ The purchase date is of the predefined XML type `date`.

- ⑥ The rating is an empty element with an attribute `stars` indicating the number of stars, which must be a number greater than 0.
- ⑦ This attribute is the link between the wines defined by the quantities in the cellar and their description in the catalog. Being of type `IDREF`, it must make reference to an existing `ID` in the catalog.
- ⑧ The element `name` is a list of elements comprising the first name, the family name and initial in any order.
- ⑩ Each component of a name is a string.
- ⑨ As all elements are optional (`minOccurs="0"`), we obtain the usual definition of a name, contrarily to what we had to do with a DTD line 12-⑥ of Example 3.1.
- ⑪ A cellar book is a list of elements starting with the wine catalog, followed by the `owner` element, an optional address and finally the description of the cellar itself using the element defined at line 8-②.
- ⑫ An address is sequence of four elements indicating the street address, city, the province in Canada and a Canadian postal code. An address is not an element but a group that can be used to define other elements. Here it is used within the `location` element of the `cellar-book` (line 42-⑩) and in the `Owner` type (line 84-⑮).
- ⑬ A postal code is a string matching a regular expression defined in the `pattern` element in the simple type defined at line 91-⑯.
- ⑭ The province is indicated by a two-letter code chosen among a predefined list.
- ⑮ An owner is a name element followed by the content of the `address` group, defined at line 57-⑫.
- ⑯ A Canadian postal code is composed of two groups of three characters alternating between a capital letter and a number. The two groups are separated by a space.

Figure 3.2. Graphical view of the schema for the wine catalog



Graphical view of the schema for the wine catalog (Example 3.4). See the caption of Figure 3.1 for an explanation of the symbols used in the figure.

Example 3.4. [wineCatalog.xsd] XML Schema file to validate the instance document shown in Example 2.3.

This file can be compared with the DTD shown in Example 3.2.

```

1 <?xml version="1.0" encoding="UTF-8"?>
  <xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'
    elementFormDefault="qualified"
    attributeFormDefault="unqualified"
5  xmlns:cat      ="http://www.iro.umontreal.ca/lapalme/wine-catalog"
    targetNamespace="http://www.iro.umontreal.ca/lapalme/wine-catalog">④

  <!-- needed because this schema will be imported -->
  <xs:import namespace="http://www.w3.org/XML/1998/namespace"
10  schemaLocation="http://www.w3.org/2001/xml.xsd"/>

  <xs:element name="wine-catalog">
    <xs:complexType>

```

```
15         <xs:sequence minOccurs="0" maxOccurs="unbounded">
            <xs:element name="wine" type="cat:Wine"/>
        </xs:sequence>
        <!-- needed because this schema will be imported...-->
        <xs:attribute ref="xml:base"/>
    </xs:complexType>
20 </xs:element>

<xs:complexType name="Wine"> 6
    <xs:sequence>
        <xs:element name="properties" type="cat:Properties"/>
25     <xs:element name="origin" type="cat:Origin"/>
        <xs:choice minOccurs="0" maxOccurs="unbounded">
            <xs:element name="tasting-note"
                type="cat:Comment" minOccurs="0"/>
            <xs:element name="food-pairing"
30                 type="cat:Comment" minOccurs="0"/>
            <xs:element name="comment"
                type="cat:Comment" minOccurs="0"/>
        </xs:choice>
        <xs:element name="price" type="xs:decimal" ></xs:element>
35     <xs:element name="year" type="xs:gYear"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="appellation" type="xs:string"/>
    <xs:attribute name="classification" type="xs:string"/>
40     <xs:attribute name="code" type="xs:ID"/> 7
    <xs:attribute name="format" type="cat:Format"/>
</xs:complexType>

<xs:complexType name="Properties"> 8
45     <xs:sequence>
        <xs:element name="color" type="cat:Color"/>
        <xs:element name="alcoholic-strength" type="cat:Percentage"/>
        <xs:element name="nature" type="xs:string" minOccurs="0"/>
    </xs:sequence>
50 </xs:complexType>

<xs:complexType name="Origin"> 9
    <xs:sequence>
        <xs:element name="country" type="xs:string"/>
55     <xs:element name="region" type="xs:string"/>
        <xs:element name="producer" type="xs:string"/>
    </xs:sequence>
</xs:complexType>

60 <xs:simpleType name="Format"> 10
    <xs:restriction base="xs:string">
        <xs:enumeration value="375ml"/>
        <xs:enumeration value="750ml"/>
        <xs:enumeration value="1l"/>
65     <xs:enumeration value="magnum">
        <xs:annotation>
            <xs:documentation> 1.5 litres</xs:documentation>
        </xs:annotation>
    </xs:restriction>
</xs:simpleType>
```



```

        </xs:annotation>
    </xs:enumeration>
70 <xs:enumeration value="jeroboam">
        <xs:annotation>
            <xs:documentation> 3 litres</xs:documentation>
        </xs:annotation>
    </xs:enumeration>
75 <xs:enumeration value="rehoboam">
        <xs:annotation>
            <xs:documentation> 4.5 litres</xs:documentation>
        </xs:annotation>
    </xs:enumeration>
80 <xs:enumeration value="mathusalem">
        <xs:annotation>
            <xs:documentation> 6 litres</xs:documentation>
        </xs:annotation>
    </xs:enumeration>
85 <xs:enumeration value="salmanazar">
        <xs:annotation>
            <xs:documentation> 9 litres</xs:documentation>
        </xs:annotation>
    </xs:enumeration>
90 <xs:enumeration value="balthazar">
        <xs:annotation>
            <xs:documentation>12 litres</xs:documentation>
        </xs:annotation>
    </xs:enumeration>
95 <xs:enumeration value="nabuchodonosor">
        <xs:annotation>
            <xs:documentation>15 litres</xs:documentation>
        </xs:annotation>
    </xs:enumeration>
100 </xs:restriction>
    </xs:simpleType>

    <xs:complexType name="Comment" mixed="true">
        <xs:sequence minOccurs="0" maxOccurs="unbounded">
105 <xs:choice>
            <xs:element name="emph" type="xs:string"/>
            <xs:element name="bold" type="xs:string"/>
        </xs:choice>
    </xs:sequence>
110 </xs:complexType>

    <xs:simpleType name="Color">
        <xs:restriction base="xs:string">
            <xs:enumeration value="red"/>
115 <xs:enumeration value="white"/>
            <xs:enumeration value="rosé"/>
        </xs:restriction>
    </xs:simpleType>

120 <xs:simpleType name="Percentage">
        <xs:restriction base="xs:decimal">

```

11

12

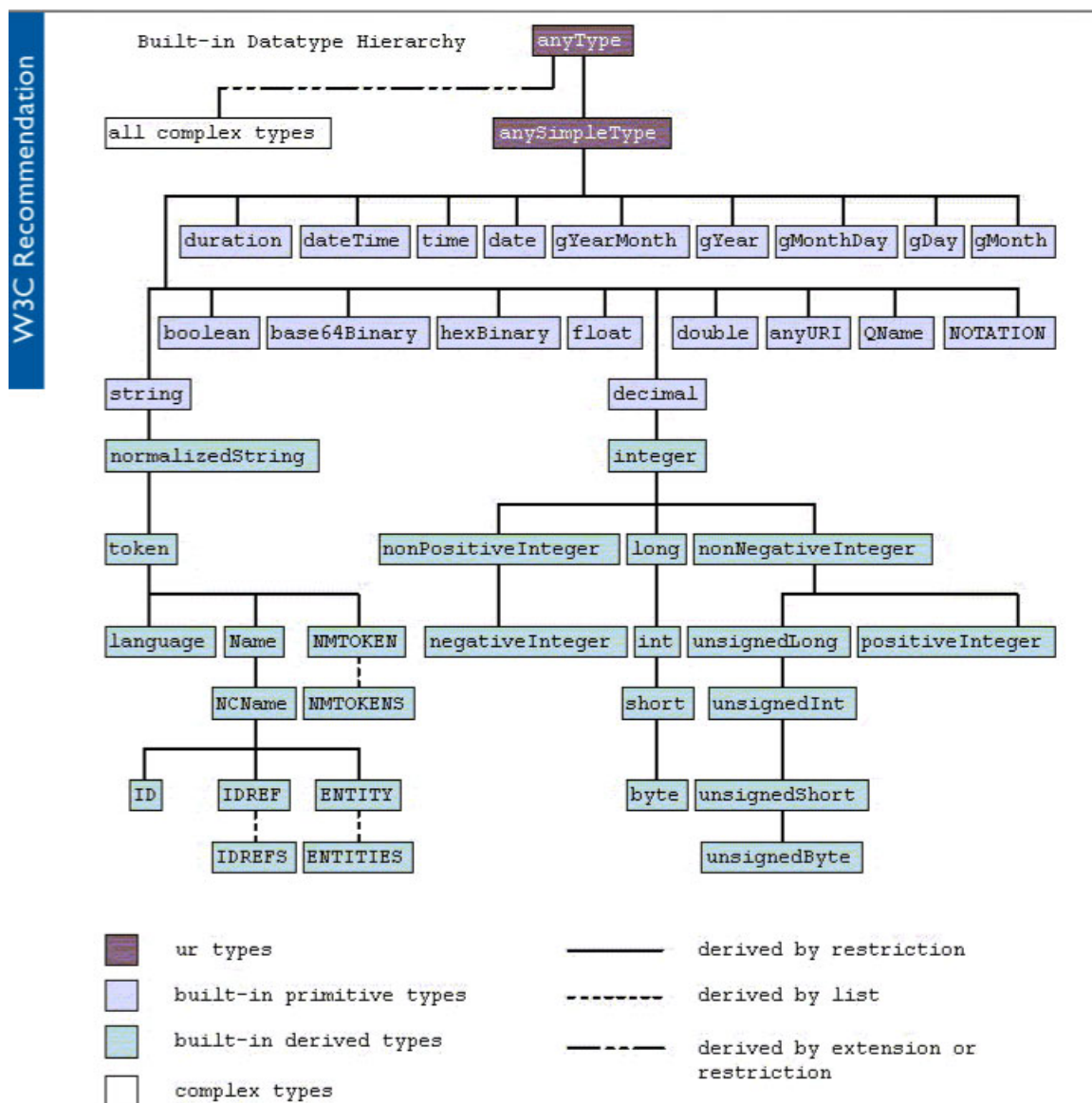
13

```
        <xs:minInclusive value="0"/>
        <xs:maxInclusive value="100"/>
        <xs:fractionDigits value="2"/>
125     </xs:restriction>
       </xs:simpleType>
</xs:schema>
```

- ❶ Start of the schema for the wine catalog giving `xs` prefix for the elements used in the definition of other elements.
- ❷ If a schema has a target namespace such as in line 6-❹, `qualified` means that all defined elements in this schema will be created in the target namespace.
- ❸ `unqualified` means that all attributes in this schema will belong to the same namespace as their carrying element.
- ❹ Defines the namespace that will be used for the elements defined by this schema.
- ❺ The wine catalog is a (possibly empty) sequence of wine definitions. In order that references to internal elements remain valid when the instance file validated by this schema is imported by another file, an `xml:base` attribute is added by the import process. This implies that this attribute must also be allowed in the schema.
- ❻ A wine entry of the catalog is a list of properties: its origins followed by optional tasting notes, food pairing suggestions and comments, its price and its production year. Other information such as the name, the appellation, the classification, the code and the format are given as attribute strings.
- ❼ A wine in the catalog is assigned a unique code of type `ID`. The validation process will ensure their uniqueness across the whole file. It will be references by the code attribute in the wine definition of the cellar.
- ❽ `Properties` defines the color of the wine, its alcoholic content as a percentage defined by a type on line 120-❿. Finally, an optional nature can be given as a string.
- ❾ `Origin` is defined by the country, the region and the producer.
- ❿ `Format` is a string value taken within the values appearing in an `xs:enumeration` element within the `xs:restriction` element. The content of the `xs:annotation` is for structured comments without any impact on the type constraints but they are kept within the schema structure. This information can then be used in XML applications, for example it can be displayed in by XML editors.
- ⓫ The `mixed` attribute having a `yes` value indicates that enclosed elements can be interleaved with text nodes. In this case, it means that the content of a comment is text with embedded `emph` and `bold` elements.
- ⓬ `color` can be only be one of three most often encountered possibilities. Wine connoisseurs might be *shocked* that wines cannot take other colors such as *yellow*.
- ⓭ A percentage is a number between 0 and 100 that must include 2 digits after the decimal point.

3.2.1. Simple Types

Figure 3.3. Built-in datatypes for XML Schema



Built-in datatypes for XML Schema. *ur*-types serve as root of the type hierarchy for all derivations. *ur* is the German prefix meaning *ancestral* such as in *Ursprung* (beginning). Figure taken from section 3 of XML Schema Part 2: Datatypes [9].

A simple type is a primitive datatype, roughly corresponding to PCDATA in DTD, such as `xs:string`, `xs:decimal`, `xs:double`, `xs:date` (XML has 19 of them, shown in Figure 3.3) or a derivation of a primitive datatype. A derivation is a restriction on the original type such as constraining the maximum length of a string, giving a list of acceptable values, or requiring that the value matches a regular expression. Figure 3.3 shows a number of built-in derived types and types that are derived from them (i.e. appear under

them). We can see uses of simple types in Example 3.3: `stars` (line 22-⑥) which must not only be an integer but a positive one, `first` (line 34-⑩) which is a `string` (essentially the same thing as a `#PCDATA` in a DTD).

Users can also define their own simple types, which cannot have any internal element, using the `xs:simpleType` element. A simple type definition can constrain a string to be one of many choices (`ProvinceCA` on line 72-⑭) or to match a regular expression (`PostalCodeCA` on line 91-⑯).

A new simple type can also be created using a `list` (allowing a series of primitive type values) or a `union` (allowing one of many primitive types). It is thus possible to define a whole gamut of types. These are quite straightforward, although the *the devil is in the details* described in [45] and [9].

3.2.2. Complex Types

Unlike a simple type, a complex type can contain element declarations, element references and attributes declarations. We will illustrate some of these possibilities with Example 3.3.

An element declaration is done with an `xs:element` specifying the name of the element and its type, which can either be defined as the value of the element, such as `cellar` (line 8-②), or by indicating the type with the `type` attribute such as in `wine` (line 11-③) or `purchaseDate` (line 18-⑤).

A complex type is defined either by a sequence of elements contained in an `xs:sequence` element (see `cellar` on line 8-②) or by a choice between many elements contained in an `xs:choice` element such as within `name` (line 31-⑧). Attributes are defined *after* the definitions of the elements in sequence or in choice even though they appear in the start-tag (see `code` as attribute of `wine`, line 27-⑦).

`xs:choice` and `xs:sequence` can be nested. For example, `name` (line 31-⑧) indicates a choice between three elements of type `string` `first`, `family` and `initial`, which can be repeated any number of times. Indeed, because an element only occurs once by default (i.e. `minOccurs="1"`) and that `maxOccurs="unbounded"`, each element can appear as often as we wish.

An existing element can also be referred to using the `ref` attribute like `name` used in `Owner` (line 84-⑮). But be aware that, in this case, if you had mistakenly used the `name` attribute instead of `ref`, you would have named a new element with no connection with the one you wished to have referenced; this can lead to errors that are difficult to track down (I learned this the hard way!!!).

In Example 3.4, the `mixed="true"` attribute in the definition of a type (see `Comment`, line 103-⑩) means that character data can also appear between the elements described by the content of the type. In this case, character data can thus be interspersed with any number of `emph` and `bold` elements.

3.2.3. Namespaces in Schemas

We have introduced namespaces for instance documents in Section 2.1, but they only show their full power during the validation process in which the combination of element names and namespaces must correspond between the instance and the schema. Of course, namespaces must be properly combined during file inclusion and details can become quite intricate.

We will illustrate a simple but quite frequent case with Example 3.3 and Example 3.4. These schemas both define a `wine` element, but with meanings and contents which must be distinguished. This is achieved with

namespace declarations. A similar name clash occurs when we must define an element called `type` or `sequence` that is already used in the schema vocabulary. This is why we define a namespace (usually `xs` or `xsd`) for the element names of an XML Schema.

By default, names without prefixes are defined in the *empty namespace* or the namespace assigned to the `xmlns` attribute. To create elements in a specific namespace (and not the empty one), we set a value for the attribute as it is done for element `targetNamespace` in Example 3.4 (line 6-④). The same namespace is also assigned to the prefix `cat`. `elementFormDefault` is set to `qualified` (line 3-② of Example 3.4) so that global elements and types of an included file are visible in the including file. The `attributeFormDefault` is set to `unqualified` to ensure that the attributes are in the same namespace as the containing element.

The import of elements from an external schema file along with its namespaces is done using `xs:import` (line 5-① of Example 3.3) indicating the namespace used in this file for the target namespace of the imported file (here we keep the same one) and the location of the file to be imported. The imported namespace must be given a prefix definition with an `xmlns` declaration, see the root tag of Example 3.3. Because the name associated with the target namespace of the imported file is the same as the one associated with the `cat` prefix, we use `cat:wine` to refer to the `wine` element of Example 3.4. Namespace and importation of RELAX NG schemas, described in the next section, are similar in principle to what we have shown for the importation of XML Schema.

We now better understand how namespaces are used in the *instance documents* and how they are linked to their schemas. For example, the first lines of Example 2.3 define the namespace associated with the *null prefix* (i.e. only the element name) as the value of the `xmlns` attribute. We also indicate the namespace and the location of the Schema to be used for validation as the value of the `xsi:schemaLocation`. The `xsi` prefix must also be defined by an attribute starting with `xmlns:`. Because all elements defined in this file are in the same package, to which we have assigned the null prefix, no namespace prefix is used in this file.

3.2.4. Overview of the XML Schemas

Example 2.1 (page 36) shows the outline of our XML instance files validated by the two XML Schemas described in this section. Their outline is shown in Example 3.5. The instance file of the wine catalog is included in the one of the cellar book. So it was natural and convenient to have a similar organization for the corresponding schemas even though this is not always the case. In this case, the wine catalog schema is included in the cellar book schema. Note the use of the namespace prefixes in both the XML instance and the corresponding XML Schema files. The part in bold characters in Example 2.1 and Example 3.5 belong to different namespaces. Here, we kept the same name for the namespaces in both the instance file of the wine catalog and the corresponding schema. In Example 3.5, we see that references to lines in bold need to use the namespace prefix `cat` for the `SAQ-code` type (line 37-⑨) and the `wine-cataloge`lement (line 43-⑩).

These examples show another interesting use of namespaces: to make sure that relative references are kept, an `xml:base` attribute (line 19-④) is added to the root element (line 14-③) by when an instance file is included into another. In order that the resulting instance file remains valid, `xml:base` must be added as an attribute in the `WineCatalog.xsd` schema. `xml:base` is itself a special XML type whose definition must also be imported (line 11-② of Example 3.5).

Example 3.5. Outline of `CellarBook.xsd` importing Example 3.4 (shown in italics) in a different namespace.

```

1 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:cat="http://www.iro.umontreal.ca/lapalme/wine-catalog">

  <xs:schema
5     xmlns:xs='http://www.w3.org/2001/XMLSchema'
     elementFormDefault="qualified"
     attributeFormDefault="unqualified"
     xmlns:cat      ="http://www.iro.umontreal.ca/lapalme/wine-catalog"
     targetNamespace="http://www.iro.umontreal.ca/lapalme/wine-catalog"> ❶

10     <xs:import namespace="http://www.w3.org/XML/1998/namespace" ❷
        schemaLocation="http://www.w3.org/2001/03/xml.xsd"/>

        <xs:element name="wine-catalog"> ❸
15           <xs:complexType>
             <xs:sequence minOccurs="0" maxOccurs="unbounded">
               <xs:element name="wine" type="cat:Wine"/>
             </xs:sequence>
               <xs:attribute ref="xml:base"/> ❹
20           </xs:complexType>
         </xs:element>

           <xs:complexType name="Wine">...</xs:complexType> ❺
           ...
25 </xs:schema>

  <xs:element name="cellar"> ❻
     <xs:complexType>
       <xs:sequence minOccurs="0" maxOccurs="unbounded">
30         <xs:element name="wine" type="Wine"/> ❼
       </xs:sequence>
     </xs:complexType>
  </xs:element>

35 <xs:complexType name="Wine"> ❽
   <xs:sequence>...</xs:sequence>
   <xs:attribute name="code" type="xs:IDREF" use="required"/> ❾
 </xs:complexType>

40 <xs:element name="cellar-book"> ❿
   <xs:complexType>
     <xs:sequence>
       <xs:element ref="cat:wine-catalog"/> ⓫
       <xs:element name="owner" type="Owner"/>
45       <xs:element name="location" minOccurs="0">...</xs:element>
       <xs:element ref="cellar"/>
     </xs:sequence>
   </xs:complexType>
 </xs:element>

```

```
50      ...  
    </xs:schema>
```

- ❶ The target namespace of the imported schema is the one specified by the `cat` prefix.
- ❷ Imports another schema whose content is shown here in italics.
- ❸ The wine catalog is a list of wine descriptions.
- ❹ Ensures that the `base` attribute added by the import process is properly defined.
- ❺ Type for the wine description in the catalog.
- ❻ Definition of the cellar in the top-level schema.
- ❼ wine elements in the cellar refer to the `Wine` type definition of the cellar.
- ❽ Type for the wine description in the cellar.
- ❾ Reference to the imported type for the wine code.
- ❿ Top-level element of the cellar.
- ⓫ Reference to the imported type for the wine-catalog (line 14-❸).

3.3. RELAX NG

As we have seen in the previous section, XML Schema allows a thorough validation of XML instance files. The type extension mechanism is very powerful but its XML format is not user-friendly, especially for complex embedding of sequences and choices. This is why the graphical editing of schemas, provided by editors such as XMLSpy and `<oxygen/>`, is very useful. In fact when it comes to ease of use, the DTD grammar-like format is much more convenient. In order to get the best of both worlds, an alternative Schema notation has been suggested which is called RELAX NG (*REgular LAnguage for XML, New Generation*). It features a simpler, intuitive notation to define schemas. RELAX NG is based on the same mathematical theory underlying regular expressions but adapted to the XML context. The mathematical foundations are both simpler and more powerful than the ones of the XML Schema.

RELAX NG has two equivalent syntaxes: one is XML-based and the other (called compact) is more convenient because it allows grammar-like definitions. Eric van der Vlist[59] has written an excellent book explaining both notations in detail. First, he introduces the XML patterns, the theoretical foundations of the formalism, that are combined into ordered and unordered groups and used in choices among alternatives. He then shows how the compact notation can simplify XML notation. In this report, we use the compact notation to develop the RELAX NG schemas. Most validators can deal directly with the compact notation. Example 3.6, a RELAX NG compact notation schema for our cellar book, looks more intuitive than the equivalent XML Schema of Example 3.3. As can be seen in Table 3.3, the structure of RELAX NG Compact definitions is quite regular and simple: a definition is simply a name followed by an equal sign and a pattern definition.

The Trang automatic Schema converter [16] can be used to obtain an XML version; the associated web site of this book shows the resulting RELAX NG version corresponding to the RELAX NG Compact given here.

A pattern can start either with the keyword `element` or `attribute` followed by another pattern within braces. Patterns can be combined sequentially (with a comma), with alternatives (with a vertical bar) or by interleaving (with an ampersand). This last case means that all patterns must occur but not necessarily in order. A pattern can also be qualified as optional, appear zero or more times or once or more. Mixed patterns allow text elements to appear between patterns. A reference to another pattern is indicated by simply giving its name (*id*). `empty` means that the content of the element must be empty. `text` corresponds to any number of text nodes in the instance document. Specifying a value (usually within braces) means that the element in the document should match this value. It is also possible to specify facets (in the XML Schema sense) to a type with a list of triples of the form: the name of the facet, an equal sign and then the value of the facet.

In Example 3.6, we can see examples of element definitions (line 8-⑤, line 18-⑦ and line 24-⑧). A definition can also be a comma-separated sequence of patterns (line 31-⑨ and line 37-⑩). We use it here for type definitions but the concept is more general and can be applied to any kind of definition. The content of a definition starts with the keyword `attribute` or `element` followed by its name and the type of its content between braces. Similarly to the regular expressions conventions described in Table B.1. If the element separator is `&` (such as for `name-element`), it indicates an *interleave*, meaning that elements in the pattern are unordered. In this case, it means that the parts of the name can appear in any order, any of them being optional because they are followed by a `?`. This is a slight difference from the syntax allowed for a name element as defined by the DTD (Example 3.1) and XML Schema (Example 3.3), in which the only way to indicate this constraint would be to enumerate all possible orderings of `first`, `family` and `initial`. The root element of the schema is defined by the rule associated with the `start` keyword.

Table 3.3. RELAX NG Compact syntax reminder

Compact Syntax (RNC)	XML Syntax (RNG)
<code>{default? namespace id=URI datatypes id=URI}* { start=pattern id=pattern }*</code>	<code><grammar> {<start> pattern </start> <define name="NCName">pattern+</define>}* </grammar></code>
Patterns	
<code>element QName «{ » pattern «}»</code>	<code><element name="QName">pattern+</element></code>
<code>attribute QName «{ » pattern «}»</code>	<code><attribute name="QName">pattern+</attribute></code>
<code>pattern{« , » pattern}+</code>	<code><group name="QName">pattern+</group></code>
<code>pattern{«&» pattern}+</code>	<code><interleave name="QName">pattern+</interleave></code>
<code>pattern{« » pattern}+</code>	<code><choice name="QName">pattern+</choice></code>
<code>pattern«?»</code>	<code><optional name="QName">pattern+</optional></code>
<code>pattern«*»</code>	<code><zeroOrMore name="QName">pattern+</zeroOrMore></code>
<code>pattern«+»</code>	<code><oneOrMore name="QName">pattern+</oneOrMore></code>
<code>mixed «{ » pattern «}»</code>	<code><mixed name="QName">pattern+</mixed></code>
<code>id</code>	<code><ref name="NCName" /></code>
<code>empty</code>	<code><empty/></code>
<code>text</code>	<code><text/></code>
<code>data TypeValue</code>	<code><value {name="NCName"}?>string+</value></code>
<code>data TypeValue «{»{id=value}* «}»</code>	<code><data {type="NCName"}?> {<param name="NCName">string</param>}* </data></code>

Reminder of the RELAX NG Compact and RELAX NG syntax used in our examples. The top cells of the table specify the start of the file for RELAX NG Compact and the root element for RELAX NG. Each line of the bottom part of the table is a different pattern that can be combined almost freely with the others. The corresponding RELAX NG Compact and RELAX NG elements appear on the same line within the bottom cell of the table. Like we did in Table 3.1, we put chevrons around terminal symbols that appear in the regular expression syntax of RELAX NG Compact. The chevrons should not appear in the RELAX NG Compact grammar. In RELAX NG Compact, `dataTypeName` can be `NCName`, `string` or `token`.

When there are no constraints on the string inside an element, then the type is `text`, but it can also refer to the built-in data types of XML Schema (see `wine`). Restrictions can also be added on types by surrounding them with braces: patterns (see `PostalCodeCA`) or enumerations (see `province` element in `Address`).

Example 3.6 includes (line 4-❶) the definitions of the wine catalog in a separate file (Example 3.7). Because the included file also has a `start` symbol, we override its definition by the definition in braces after the name of the file. Any other included definition could be overridden in this way. There are many other possibilities to combine definitions of many files but we will not present them in this document. One should consult [59] (Chapter 10) for more details.

Namespace prefixes are declared by a definition following the keyword `namespace` (line 1-❶). To use the predefined types of XML Schema (Figure 3.3), we similarly declare the prefix used for referring to them. RELAX NG does not implement the notions of XML Schema `keys` and `keyref` so that one must resort to the simpler (but often sufficient) notion of DTD `ID` and `IDREF` explained in Section 3.1.

Example 3.6. [CellarBook.rnc] RELAX NG Compact schema for the cellar book to validate Example 2.2.

Compare this file with Example 3.3.

```

1 namespace cat = "http://www.iro.umontreal.ca/lapalme/wine-catalog" ❶
  datatypes xs = "http://www.w3.org/2001/XMLSchema-datatypes"      ❷

  include "WineCatalog.rnc" {                                       ❸
5    start = cellar-book                                           ❹
  }

  cellar = element cellar {                                         ❺
    element wine {                                                 ❻
10      attribute code {xs:IDREF},
        element purchaseDate {xs:date},
        element quantity {xs:nonNegativeInteger},
        element rating {attribute stars {xs:positiveInteger}?},
        element comment {Comment}?
15    }*
  }

  name = element name {                                           ❼
20      element first {text}?
        & element family {text}?
        & element initial {text}?
  }

  cellar-book = element cellar-book {                               ❽
25      wine-catalog,
        element owner {Owner},
        element location {Address}?,
        cellar
30    }

  Address = element street {text},                                  ❾
            element city {text},
            element province {"AB"|"BC"|"MB"|"NB"|"NL"|"NT"|"
                               "NS"|"NU"|"ON"|"QC"|"SK"|"YT"},
35    element postal-code {PostalCodeCA}

  Owner = name, Address                                           ❿

  PostalCodeCA = xs:string {pattern="[A-Z][0-9][A-Z] [0-9][A-Z][0-9]"} ❶⓫
40

```

- ❶ Defines the namespace prefix used in the catalog.
- ❷ Defines the namespace prefix corresponding to the standard XML datatypes.
- ❸ Includes the schema for the wine catalog.
- ❹ Defines the start symbol for this file by redefining the start symbol imported from the wine catalog file.
- ❺ The cellar element is a possibly empty list of wine elements.
- ❻ A wine element is composed of a mandatory code attribute, a date of purchase, a quantity given by a non-negative integer, an optional rating (an empty element with an attribute indicating the number of stars) and finally an optional comment.

- ⑦ A name is an interleaving of a first name, family name and initial (all of them being optional). This means that these three elements can occur in any order but not more than once. Interleaving is a of expressing that elements can appear in any order. Its use here corresponds to the `xsd:all` element but in fact, it is more general and allows the unordered combination of any subgroups of elements.
- ⑧ A `cellar-book` element contains the `wine-catalog`, a description of the owner, the optional location of the cellar and the `cellar-element` itself defined on line 8-⑥. `Owner` and `Address` are themselves patterns.
- ⑨ Pattern for the street, the city name, a Canadian province name (the list of available values that can be defined directly).
- ⑩ A pattern combining an element and a pattern.
- ⑪ A pattern corresponding to a standard type with a constraint expressed as a regular expression. Here it corresponds to the Canadian postal codes.

The beginning of Example 3.7 illustrates how to declare a default namespace for the elements of this file, included in Example 3.6 (line 4-④). The definition of elements follows the same principles explained for the cellar book. `wine-catalog` must add an optional attribute `xml:base` that is used by the XML processor during the file inclusion process. It is needed in order to ensure the integrity of both the including and included file. Element `Format` shows that comments starts with a `#` and go up to the end of the line.

Example 3.7. [WineCatalog.rnc] RELAX NG Compact schema for the wine catalog to validate Example 2.3

This file can be compared with Example 3.4.

```

1 default namespace = "http://www.iro.umontreal.ca/lapalme/wine-catalog"
  datatypes xs = "http://www.w3.org/2001/XMLSchema-datatypes"

  start = wine-catalog
5
  wine-catalog = element wine-catalog {
                                # needed because this schema will be imported
                                attribute xml:base{text}?,
                                wine*
10                                }

  wine = element wine{
                                attribute name {text},
                                attribute appellation {text},
15                                attribute classification {text},
                                attribute code {xs:ID},
                                attribute format {Format},
                                properties,
                                origin,
20                                ( element tasting-note {Comment}
                                  | element food-pairing {Comment}
                                  | comment
                                )*,
                                element price {xs:decimal},
25                                element year {xs:gYear}

```

```

    }

    properties = element properties {
        element color {Color},
30         element alcoholic-strength {Percentage},
        element nature {text}?
    }

    origin = element origin {
35         element country {text},
        element region {text},
        element producer {text}
    }

40 Format = "375ml" | "750ml" | "1l"
        | "magnum" # 1.5 litres
        | "jeroboam" # 3 litres
        | "rehoboam" # 4.5 litres
45         | "mathusalem" # 6 litres
        | "salmanazar" # 9 litres
        | "balthazar" # 12 litres
        | "nabuchodonosor" # 15 litres

    Comment = mixed {element emph {text}* & element bold {text}*}
50 comment = element comment{Comment}

    Color = "red" | "white" | "rosé"

    Percentage = xs:decimal {
55         minInclusive = "0"
        maxInclusive = "100"
        fractionDigits = "2"}

```

- ❶ The wine catalog is a list of wine elements. An `xml:base` attribute is added to schema to take into account that it will be added during the importing process.
- ❷ A pattern defining all the information related to a given wine.
- ❸ A pattern describing some properties of a wine.
- ❹ Informations about the origin of a wine.
- ❺ List of allowable values for wine format.
- ❻ A comment is text interspersed with `emph` and `bold` elements which also contain text. Note that this definition does not allow embedding of `emph` and `bold` elements.
- ❼ List of allowable values for the color of wines.
- ❽ A percentage based on standard XML types and constraint attributes.

3.4. Schematron

In the previous sections, we have seen two different ways of validating the content of an XML file: while DTDs can only validate the element nesting structure, XML and RELAX NG schemas can also validate the content of the elements by enforcing local constraints such as the length of strings, regular expressions or range on the allowed values. These types of constraints, defined by means of grammars described the well-formedness of structure and values, can be characterized as *syntactic*. But in typical applications, more *semantic* types of constraints could be appropriate for checking long distance dependencies that would be awkward or even impossible to express by means of grammar rules or regular expressions, e.g. a starting date in an attribute or element should be earlier than an ending date in another one. In this case, a rule-based approach is more appropriate.

Typical global semantic constraints can be defined to take into account relations between elements and attributes: for example, an attribute with a given value could imply the definition of another element. It could be necessary that a certain value is equal to the sum of other values. In certain cases, it would even be interesting to enforce constraints between different XML files. These checks are outside the scope of the XML validation methods we have presented in the previous sections, but an alternative approach, called Schematron [20], [86] has been developed to cope with this type of validation. Instead being grammar-based, Schematron is a rule-based validation approach to define assertions in which XPath expressions are evaluated. XPath expressions will be described more fully in Chapter 4, but for the moment they can understood as defining values computed over different parts of an XML file; in fact, Schematron even allows combining values spanning more than one XML file. In principle, Schematron could be used for also defining local constraints, but as they are much more easily defined by grammars, we will use Schematron patterns as supplementary validation rules used in conjunction with an XML Schema. The Schematron specification describes how patterns can be embedded within other XML schema notations such as XML Schema, RELAX NG and RELAX NG Compact.

Similarly to an XML Schema and RELAX NG file, a Schematron file is a well-formed (and valid) XML file that defines validation in terms of `pattern` elements containing `rule` elements. A `rule` defines a context for internal XPath expressions used in `assertion` and `report` elements. When the XPath expression of an `assertion` (resp. `report`) is `false` (resp. `true`), then the content of the element is output as validation message. The validation message can contain variable parts to customize the message according to the context and the specific elements involved.

Typical *semantic* validation uses are the following:

cooccurrence checking	defines a relationship between attributes, elements or a mix of these not only within a single XML document but also across multiple documents; it can also place constraints on the context of mixed content which cannot be enforced with a grammar-based approach.
cardinality checking	defines a constraint on the number of occurrences of data not only within a single element but even over a portion of a document.
algorithmic checking	involving some computation on values occurring in the document

Example 3.8 shows Schematron rules for the cellar book. These rules should be combined with the XML Schema or RELAX NG in order to validate the instance file both syntactically and semantically. We use the ISO Schematron [67] that has been recently standardised,. It differs in some respects from the Schematron 1.6 dialect [87], but the main ideas remain valid in both formalisms.

Table 3.4 presents the XML elements we use in our example to define the types needed for the validation of our wine catalog and cellar book.

Table 3.4. Schematron syntax reminder

<code><schema targetNamespace="URI"></code> <i>title? ns? pattern+</i> <code></schema></code>
<code><ns prefix="QName"</code> <code>uri="URI"/></code>
<code><title></code> <i>PCDATA</i> <code></title></code>
<code><pattern abstract="yes no"</code> <code>id="ID"</code> <code>is-a="IDREF"></code> <i>param* rule+</i> <code></pattern></code>
<code><param name="QName"</code> <code>value="XPathExpr"/></code>
<code><rule context="XPathExpr"></code> <i>{let report assert}*</i> <code></rule></code>
<code><let name="QName"</code> <code>value="XPathExpr"/></code>
<code><report test="XPathExpr"></code> <i>{PCDATA value-of name}*</i> <code></report></code>
<code><assert test="XPathExpr"></code> <i>{PCDATA value-of name}*</i> <code></assert></code>
<code><value-of select="XPathExpr"/></code>
<code><name/></code>

A reminder of the subset of the Schematron syntax used in Example 3.8. Names in italics refer to other elements. PCDATA refers to XML string content. Regular expressions (see Table B.1) are used to describe the allowed forms.

Example 3.8. [CellarBook.sch] ISO-Schematron for the cellar book to validate Example 2.2.

This file can be used in combination with the XML Schema shown in Example 3.3.

```

1 <?xml version="1.0" encoding="UTF-8"?>
  <schema xmlns="http://purl.oclc.org/dsdl/schematron"                ❶
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"                ❷
    queryBinding="xslt2">
5   <title>Validation using Schematron rules</title>                  ❸
   <ns prefix="cat"                                                  ❹
     uri="http://www.iro.umontreal.ca/lapalme/wine-catalog"/>
   <xsl:key name="colors" match="/cellar-book/cat:wine-catalog/cat:wine" ❺
     use="cat:properties/cat:color"/>
10
   <pattern>                                                         ❻
     <rule context="wine">
       <report test="rating/@stars>1 and not(comment)">
         There should be a comment for a wine with more than one star.

```

```

15         </report>
        </rule>
    </pattern>

    <pattern> 7
20     <rule context="cellar">
        <let name="nbBottles" value="sum(wine/quantity)"/> 8
        <report test="$nbBottles < 10">
            Only <value-of select="$nbBottles"/> bottles left in the cellar.
        </report>
25     <!-- nb of bottles of each color in the cellar -->
        <let name="winesFromCellar" value="/cellar-book/cellar/wine"/> 9
        <let name="nbReds"
            value="sum($winesFromCellar[@code=key('colors','red')/@code]/quantity)"/>
        <let name="nbWhites"
30     value="sum($winesFromCellar[@code=key('colors','white')/@code]/quantity)"/>
        <let name="nbRosés"
            value="sum($winesFromCellar[@code=key('colors','rosé')/@code]/quantity)"/>
        <let name="nbColors" value="$nbReds+$nbWhites+$nbRosés"/>
        <!-- check for a well balanced cellar!!! -->
35     <assert test="$nbReds>$nbColors div 3">
            Not enough reds (<value-of select="$nbReds"/> over
            <value-of select="$nbColors"/>) left in the cellar.
        </assert>
        <assert test="$nbWhites>$nbColors div 4">
40     Not enough whites (<value-of select="$nbWhites"/> over
            <value-of select="$nbColors"/>) left in the cellar.
        </assert>
        <assert test="$nbRosés>$nbColors div 4">
45     Not enough rosés (<value-of select="$nbRosés"/> over
            <value-of select="$nbColors"/>) left in the cellar.
        </assert>
        <!-- check for consistency within number of bottles -->
        <assert test="$nbBottles=$nbColors"> 10
            Inconsistent count of bottles: total is <value-of select="$nbBottles"/>
50     but the count by colors is <value-of select="$nbColors"/>:
            (<value-of select="$nbReds"/> reds, <value-of select="$nbWhites"/>
            whites and <value-of select="$nbRosés"/> rosés).
        </assert>
    </rule>
55 </pattern>

    <pattern abstract="true" id="spacesAtStartEnd" 11
        <rule context="comment|cat:comment|cat:food-pairing|cat:tasting-note">
            <report test="starts-with($elem,' ') or
60     substring($elem,string-length($elem))=' '>
                A <value-of select="name($elem)"/> element within a <name/>
                should not start or end with a space.
            </report>
        </rule>
65 </pattern>

    <pattern is-a="spacesAtStartEnd" 12
        <param name="elem" value="cat:bold"/>
    </pattern>

```



```

70     <pattern is-a="spacesAtStartEnd">
        <param name="elem" value="cat:emph" />
    </pattern>

</schema>

```

13

- ❶ ISO-Schematron rules are defined within an XML element defined in a specific namespace. We also declare the `xsl` namespace because it will be needed for using `xsl:key` line 8-❶ for indexing
- ❷ By setting `queryBinding` attribute to `xslt2`, it is possible to use XPath 2.0 expressions in the assertions. By default, it is limited to XPath 1.0.
- ❸ Gives an informative title to a set of validation rules.
- ❹ Defines the namespace prefix to be used within XPath expressions that refer to elements of the wine catalog (see line 8-❶). For implementation reasons, it is not enough, or even necessary, to define the namespace with a `xmlns` attribute.
- ❺ Declares that `wine` elements of the catalog are to be indexed by their `color`, used in line 26-❹.
- ❻ Defines a *cooccurrence constraint* within a `wine` element: when the `rating` is more than one star then there should be a `comment` element. The rule element gives the context and the content of the `report` element defines the message which is output when the XPath expression of the `test` attribute evaluates to `true`.
- ❼ A pattern that defines a rule implementing a mix of *cooccurrence and algorithmic constraint* within a `cellar` element.
- ❽ There should be a warning when the total number of bottles in the cellar is less than 10. This is obtained by summing the values of all the `quantity` elements of the cellar. We save the total in the local variable `nbBottles` and use its value in the test by prefixing its name by `$`. The message can be customized by embedding `value-of` elements which are replaced by the result of the evaluation of the `select` attribute.
- ❾ Defines a mix of cooccurrence and algorithmic constraints by computing the total number of bottles of each color and asserting that there should be at least a third of the number of bottles that are red, one fourth of both rosés and whites. The codes of wines of a given color are found by searching with the key XPath function. An `assert` element differs from a `report` in that the message is output when the test expression is `false`. We define a list of local variables which can be used in many assertions within the same context.
- ❿ Checks that the total number of bottles in the cellar is equal to the sum of the numbers of each color. This an *internal* consistency check which in principle should never appear because it would mean that there a wines that are not red, white or rosé. But these values are the only ones allowed by the XML Schema
- ⓫ Defines an *abstract* pattern for generating two other patterns (line 66-⓫, line 69-⓫). This abstract pattern can be applied to `$elem` elements in many contexts; it makes sure that the string content of these elements do not start or end by a space. This is an example of the kind of validation possible on mixed content on which XML Schema place few constraints.
- ⓬ Instantiates the abstract pattern defined at line 57-⓫ for the `cat:bold` elements.
- ⓭ Instantiates the abstract pattern defined at line 57-⓫ for the `cat:emph` elements.

This example shows how new kinds of validation rules can be applied. The main advantages being the possibility of enforcing long-distance or algorithmic dependencies that cannot be implemented by the grammar-based validation offered by DTD and XML Schemas. Another advantage is the possibility of giving more meaningful error messages for the user than the ones generated automatically by a grammar based validation. As these validations are enforced by run-time checking once the user has entered the values in the file, it is not possible to determine a list of allowed choices that can be displayed by an XML editor

at any point in the file. Although the ISO specification is relatively short (about 20 pages), we have not covered here all aspects of Schematron. In particular, we have not described the order of evaluation of the patterns and rules and we omitted some elements that are used for formatting and organizing processing phases and diagnostic messages, even multi-lingual ones.

In principle, Schematron could be used as the sole validation mechanism for XML instance files but simple sequencing constraints that are easy to express in a grammar must rely on complex XPath expressions involving `::following-sibling[1]`. So in practice, it is more convenient to use both types of validation. But if having two separate validation files is not suitable, it is possible to embed Schematron rules within `xs:appinfo` and `xs:annotation` elements of an XML Schema. Similar embedding conventions have been defined for both RELAX NG and RELAX NG Compact.

3.5. Associating an Instance File with a Schema

An instance XML file can specify its validating schema by adding some information in the attributes of the root tag. This is illustrated in line 8-⑤ of Example 2.2 (page 10), where we indicate the location of the schema with no namespace using the `xsi:noNamespaceSchemaLocation` attribute. We then include (using an `xi:include` element) the `WineCatalog.xml` file (Example 2.3) so that its elements can be referred to. In fact, the XML processor sees the full content of these files (i.e. the cellar and the wine catalog). Example 2.1 ((page 9)) illustrates the file inclusion mechanism for the instance files. They correspond to their respective XML Schema in Example 3.5.

`xi:include` refers to the W3C standard [35] which specifies a general purpose inclusion mechanism to merge information from different XML files. So it is possible to include only some well-formed parts of the included file, but here we include the whole wine catalog. This is a principled way of including information and not mere character inclusions like the one specified with DTD system entities used in Section 3.1.1.

Example 2.2 (page 10) also shows that even if a file is validated with an XML Schema, a `DOCTYPE` element can be added to define new entities. In fact, it is the only way to define an entity in an instance file validated with an XML Schema.

line 1-① of Example 2.3 (page 12) shows how to link an instance file and define its namespace. The empty namespace, defined by the `xmlns` attribute in the root tag (line 4-⑥), indicates that all element tags without prefix are defined in the `http://www.iro.umontreal.ca/lapalme/wine-catalog` namespace. The schema location is indicated as the value of the `xsi:schemaLocation` (line 2-②) attribute with two values (blank separated). The first part indicates the namespace corresponding to the target namespace of the schema and the second part gives its URI (here the local file `WineCatalog.xsd`).

RELAX NG specifications [15] do not prescribe how an instance file should be linked to its schema, so each XML editor or validator has an implementation-specific way of associating these files (either internally or externally). For example, `<oxygen/>` uses processing instructions inserted at the top of the file such as the following (depending on whether the compact syntax is used or not):

```
<?oxygen RNGSchema="CellarBook.rnc" type="compact"?>
<?oxygen RNGSchema="CellarBook.rng" type="xml"?>
```

The Schematron specification does not prescribe the link mechanism between an instance file and its Schematron rule base. With the <oXygen/> XML editor, it is sufficient to add a processing instruction like `<?oxygen SCHSchema="CellarBook.sch" ?>` to a XML Schema validated file such as Example 2.2 or to get **both** types of validation: XML Schema and Schematron. It is also possible to embed Schematron rules within `xs:appinfo` and `xs:annotation` elements of an XML Schema which is then linked as described above. Similar embedding conventions have been defined for both RELAX NG and RELAX NG Compact.

3.6. Additional Information on XML Schema

Although XML schemas have been standardized, the area of validation is still a research subject and alternatives have been proposed: see [29] for a comparison between some of them. Interesting links are being made with relational database models [34] in order to build on its strong theoretical background. Schemas and the validation process are being formalized [13].

We have only skimmed over the subject of validation of XML files but the same essential ideas apply throughout. On top of the official and informal information available at <http://www.w3.org/xml/Schema>, some good sources of information and interesting tutorials can be found in the following resources:

- <http://www.XML.com> is maintained by the O'Reilly editor with many excerpts from their books.
- <http://www.XML.org> is a market-oriented site with interesting files in the *resources* section.
- <http://www.muhimbi.com/Products/XML-Quick-Reference-Sheet.aspx> a very useful XML Syntax Quick Reference Sheet (US letter size).
- <http://www.xfront.com/xml-schema.html> a complete XML Schema tutorial in roughly 150 Microsoft Powerpoint slides with example source code
- <http://www.xfront.com/schematron/> Tutorial and links for Schematron
- <http://www.xmlspy.com> XMLSpy is a commercial XML editor on the Microsoft Windows platform, with a powerful structure editor and internal validation and real-time suggestions of allowable elements attributes (strangely, these suggestions are not adequate in the *text view* i.e. the mode in which XML tags are explicitly typed). It is easy to switch between the text view and the structural view of the editor. There is also a good stylesheet designer module (Stylevision) to create stylesheet transformations interactively and graphically. These transformations can then be used as a basis for what is called the *authentic* view which can effectively hide the XML tags from the user of an XML document.
- <http://www.oxygenxml.com/> <oXygen/> is an XML editor for Microsoft Windows, Linux, MacOS X and Solaris. Real-time valid suggestions are offered in the text view. Validation can be done within the editor. Stylesheets transformations can be displayed in a window of the editor. It also features table and tree editing modes and a similar graphical output of a schema to what is provided by XMLSpy. Unfortunately, it is not possible to edit the schema graphically. It is one of the few XML editors that

also offers real-time Schematron validation. For XML files, there is also an *author* mode which allows a *tagless* XML editing of the file directly from the rendered version through Cascading Style Sheets. As the formatting allowed with CSS is less powerful than what is possible with StyleSheets, this mode does not allow a full rendering like the *authentic* mode of XMLSpy but it has the advantages of being based on open standards. But for documented oriented XML files such as the ones written in DocBook or XHTML, this mode is quite convenient and user friendly.

<http://www.thaiopensource.com/nxml-mode/> nXML mode in Emacs [19] offers real-time valid suggestions for editing XML files in text mode only when their schema is written in RELAX NG. Trang can be used for translating an XML Schema or a DTD into RELAX NG.

Chapter 4. XPath

Because XML documents are tree-structured, we must be able to designate nodes in their trees either absolutely (i.e. starting from the root) or relatively to a given node. An XPath expression refers to either a single node or to a set of nodes in the document tree. In this report, we use the XPath 2.0 syntax [17], standardized in 2006 and implemented by many XSLT processors. It is more powerful and better principled than the previous XPath 1.0 standard. Therefore make sure to run the examples of this document with a processor meeting both XPath 2.0 and XSLT 2.0 standards. XPath 2.0 has become a full-blown programming language, as can be seen from the 400 pages of chapters 7 thru 13 of the book of Michael Kay [27]. Thus we only give an overview in these few pages.

In 2014, the XPath 3.0 recommendation was published with many new features, such as dynamic function call, mapping operators and a string concatenation operator, but these features are not used in this simple tutorial.

XPath is an expression language designed primarily for accessing nodes in an XML document. It cannot modify the XML document, nor does it permit the creation of new nodes in a document. An XPath expression has one of the following types of values:

- *atomic values* of any XML Schema type (see Section 3.2) either simple ones such as integers, boolean, dates, URI or even user-defined ones through a Schema.
- *trees or nodes of the document*
- *sequence* of atomic values and trees, but **not** of sequences (sequences cannot be nested)

4.1. XPath expression components

An XPath expression is composed of *steps* separated by `/`. Each step designates an *axis* which can be understood as the direction in which the nodes will be collected from the current node (called here the starting node) in the document tree. An axis is a predefined name followed by `::` (Figure 4.1 some examples of axes). The axis specification is followed by a *node test* which is often the name of a node or `*` to indicate any element. The node(s) thus specified can be further restricted with a *predicate expression* within square brackets; this is a boolean expression evaluated at each node, which is kept in the result sequence if it evaluates to `true`; it can also be a positive integer in which case the sequence only keeps the node with this index.

Using the instance document shown in Example D.1 (page 243), here are a few examples of XPath expressions:

- `/child::cellar-book/child::cellar/child::wine[1]` designates the node corresponding to the first wine, the one whose `code` attribute is `C00043125`.
- `/descendant::wine[purchaseDate > '2005-01-01']` returns a sequence of nodes designating the wines bought in 2005 or later (the ones with `code` attributes `C00043215` and `C10263859`).
- `/descendant::wine/attribute::code` returns the sequence containing the codes of the 4 wines in the cellar.
- `(/descendant::wine/attribute::code)[4]` returns the fourth wine. Note that sequences in XPath are numbered starting from 1 and not 0 as is the case in most programming languages.

This notation is very powerful but a bit tedious to write so XPath designers have defined an abbreviated syntax for the most frequent cases: `child::` can be omitted, `attribute::` can be written as `@` and

`descendant::` is written as `//`. This notation is thus similar to what is used in computer systems to designate files and directories; except that at each step (i.e. between slashes) many nodes can be selected instead of only one file or directory. The above examples can thus be *simplified* as

- `/cellar-book/cellar/wine[1]`
- `//wine[purchaseDate>'2005-01-01']`
- `//wine/@code`
- `(//wine/@code)[4]`

After this quick overview of XPath, we give some more details on the evaluation of its expressions, which is at the root of XSLT. We remind the reader of the seven types of nodes in an XML document:

root	the starting point of the document
element	the most common type of node, it may contain other elements
text	containing the <i>real</i> information; it cannot contain any element
attribute	string information contained in the start-tag of the containing element, it is considered a child of the element which contains it
comment	information that is normally ignored for processing but that is nevertheless kept in the structure of the document
processing instruction	elements starting with <code><?</code> that will not be discussed in this document
namespace	information about the namespace of an element, its processing will not be discussed in this section.

An XPath expression designating a sequence of nodes in the document tree consists of three parts.

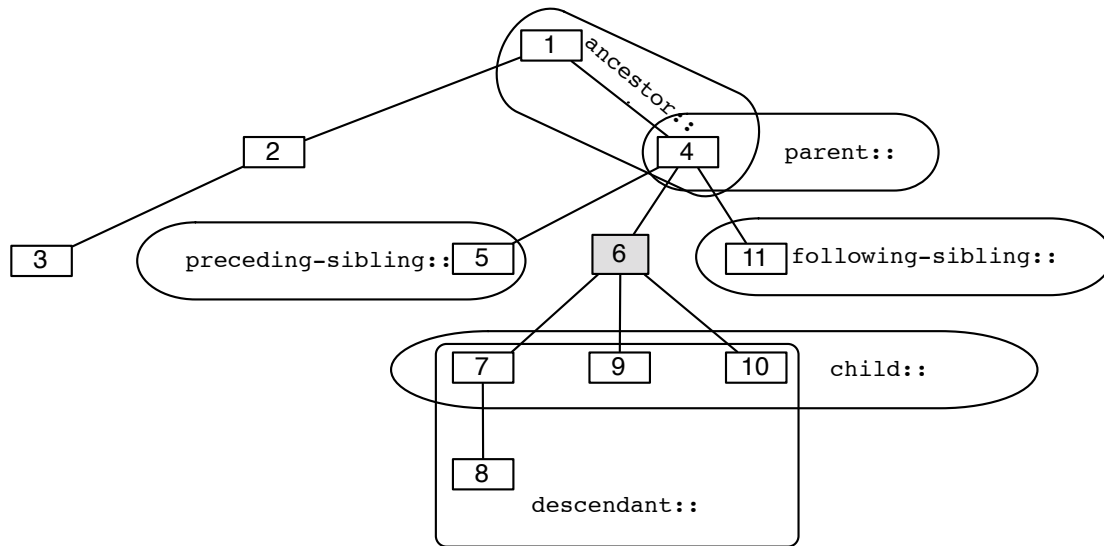
1. an *axis specifier* gives the path to a set of nodes. Most of the time, we can use the *abbreviated* syntax.
2. a *node test* can be the name of a node (with or without the namespace prefix), `*` to indicate all elements, or it can be a function name such as `node()`, `text()` or `comment()` to indicate the type of the node that is sought.
3. a *predicate* is a boolean expression given between square brackets (`[]`) that can further filter the set of nodes identified with the axis specifier and the node test. If the expression is a number i , then it refers to the i^{th} child element (numbering starts at 1).

The following grammar gives the rules of an XPath expression as taken from the XPath standard [6]. Terminals are shown in `monospace` font. As an extended BNF notation (EBNF) is used, some terminals such as parenthesis, star, plus or vertical bar are used for grouping, to indicate repetition and alternation. There are essentially the same symbols as the ones used in DTD or Relax NG compact notation. When a symbol of the EBNF also occurs as a terminal, it is enclosed within angle quotes (`« »`).

Similarly to other *functional* programming languages, XPath defines an expression in terms of a sequence of one or more expressions [rule 2]. The comma (`,`) operator is used to create a sequence out of two sequences which are *concatenated*; sequences of sequences are not allowed.

The building blocks of an XPath expressions are the following:

- *Numeric and string constants* [rules 41 and 42] their syntax is not defined here as it is conventional; as there are no boolean constants, they are created using the functions call `true()` and `false()`;
- *Variable* are referenced using the name of the variable prefixed with `$` [rule 44]. The same syntax is used for the index variable of the `for` construct;

Figure 4.1. Some XPath axes

Given a document tree rooted at node 1 and a starting node indicated by node 6 (in grey), the rounded rectangles wrap the sequence of nodes in some XPath axes. The `attribute::` axis is not shown here because it would be *within* the grey node.

- *Function calls* use the name of the function followed by the value of the actual parameters within parentheses [rule 48]. When a user-defined function is called, its name must start with its namespace prefix;
- *Range expression* [rule 11] creates a sequence of scalar values between their bounds given by two numbers;
- *Axis steps* [rules 28-39] have been described above;
- *Operators* [rules 7-24] using the standard arithmetic operations [rules 12,13] and some set-like operations [rules 14,15] on sequences; conditional expressions [rule 7] provide a compact notation for creating new values; on top of the usual comparisons [rule 22] which are applied on the values taking into account their types, there are *value* comparisons [rule 23] which makes sure that the types are compatible and *node* comparisons [rule 24] which take into account the document order of the nodes;
- *Filter expressions* [rules 39-40] are predicates, boolean expressions or a single number, enclosed in square brackets that further restrict the values produced by the previous one;
- *For expressions* [rule 4] return the sequence of results evaluating the body (expression following `return`) for each item in the subject (expression preceding `return`); an index variable is defined to reference an item of the subject sequence within the body;
- *Quantified expressions* [rule 6] are a special type of loop for checking if any or all elements of a sequence satisfy a certain predicate.

XPath 2.0 grammar

- [1] $XPath ::= Expr$
 [2] $Expr ::= ExprSingle (, ExprSingle)^*$

```

[3]      ExprSingle ::= ForExpr
           | QuantifiedExpr
           | IfExpr
           | OrExpr
[4]      ForExpr ::= SimpleForClause return ExprSingle
[5]      SimpleForClause ::= for $ VarName in ExprSingle ( , $ VarName in ExprSingle)*
[6]      QuantifiedExpr ::= (some | every) $ VarName in ExprSingle
           ( , $ VarName in ExprSingle)* satisfies ExprSingle
[7]      IfExpr ::= if «(» Expr «)» then ExprSingle else ExprSingle
[8]      OrExpr ::= AndExpr ( or AndExpr)*
[9]      AndExpr ::= ComparisonExpr ( and ComparisonExpr)*
[10]     ComparisonExpr ::= RangeExpr ( (ValueComp | GeneralComp | NodeComp) RangeExpr)?
[11]     RangeExpr ::= AdditiveExpr ( to AdditiveExpr)?
[12]     AdditiveExpr ::= MultiplicativeExpr ( («+» | -) MultiplicativeExpr)*
[13]     MultiplicativeExpr ::= UnionExpr ( («*» | div | idiv | mod) UnionExpr)*
[14]     UnionExpr ::= IntersectExceptExpr ( (union | «|» ) IntersectExceptExpr)*
[15]     IntersectExceptExpr ::= InstanceofExpr ( (intersect | except) InstanceofExpr)*
[16]     InstanceofExpr ::= TreatExpr ( instance of SequenceType)?
[17]     TreatExpr ::= CastableExpr ( treat as SequenceType)?
[18]     CastableExpr ::= CastExpr ( castable as SingleType)?
[19]     CastExpr ::= UnaryExpr ( cast as SingleType)?
[20]     UnaryExpr ::= ( - | «+»)* ValueExpr
[21]     ValueExpr ::= PathExpr
[22]     GeneralComp ::= = | != | < | <= | > | >=
[23]     ValueComp ::= eq | ne | lt | le | gt | ge
[24]     NodeComp ::= is | << | >>
[25]     PathExpr ::= (/ RelativePathExpr?)
           | (// RelativePathExpr)
           | RelativePathExpr
[26]     RelativePathExpr ::= StepExpr ((/ | //) StepExpr)*
[27]     StepExpr ::= FilterExpr | AxisStep
[28]     AxisStep ::= (ReverseStep | ForwardStep) PredicateList
[29]     ForwardStep ::= (ForwardAxis NodeTest) | AbbrevForwardStep
[30]     ForwardAxis ::= (child ::) | (descendant ::) | (attribute ::) | (self ::)
           | (descendant-or-self ::) | (following-sibling ::)
           | (following ::) | (namespace ::)
[31]     AbbrevForwardStep ::= @? NodeTest
[32]     ReverseStep ::= (ReverseAxis NodeTest) | AbbrevReverseStep
[33]     ReverseAxis ::= (parent ::) | (ancestor ::) | (preceding-sibling ::)
           | (preceding ::) | (ancestor-or-self ::)
[34]     AbbrevReverseStep ::= ..
[35]     NodeTest ::= KindTest | NameTest
[36]     NameTest ::= QName | Wildcard
[37]     Wildcard ::= «*» | (NCName : «*») | («*» : NCName)
[38]     FilterExpr ::= PrimaryExpr PredicateList
[39]     PredicateList ::= Predicate*
[40]     Predicate ::= [ Expr ]
[41]     PrimaryExpr ::= Literal | VarRef | ParenthesizedExpr | ContextItemExpr | FunctionCall
[42]     Literal ::= NumericLiteral | StringLiteral

```


[43]	<i>NumericLiteral</i> ::= IntegerLiteral DecimalLiteral DoubleLiteral
[44]	<i>VarRef</i> ::= \$ VarName
[45]	<i>VarName</i> ::= QName
[46]	<i>ParenthesizedExpr</i> ::= «(» Expr? «)»
[47]	<i>ContextItemExpr</i> ::= .
[48]	<i>FunctionCall</i> ::= QName «(» (ExprSingle (, ExprSingle)*)? «)»
[49]	<i>SingleType</i> ::= AtomicType «?»?
[50]	<i>SequenceType</i> ::= (empty-sequence «(» «)») (ItemType OccurrenceIndicator?)
[51]	<i>OccurrenceIndicator</i> ::= «?» «*» «+»
[52]	<i>ItemType</i> ::= KindTest (item «(» «)») AtomicType
[53]	<i>AtomicType</i> ::= QName
[54]	<i>KindTest</i> ::= DocumentTest ElementTest AttributeTest SchemaElementTest SchemaAttributeTest PITest CommentTest TextTest AnyKindTest
[55]	<i>AnyKindTest</i> ::= node «(» «)»
[56]	<i>DocumentTest</i> ::= document-node «(» (ElementTest SchemaElementTest)? «)»
[57]	<i>TextTest</i> ::= text «(» «)»
[58]	<i>CommentTest</i> ::= comment «(» «)»
[59]	<i>PITest</i> ::= processing-instruction «(» (NCName StringLiteral)? «)»
[60]	<i>AttributeTest</i> ::= attribute «(» (AttributeNameOrWildcard (, TypeName)?)? «)»
[61]	<i>AttributeNameOrWildcard</i> ::= AttributeName «*»
[62]	<i>SchemaAttributeTest</i> ::= schema-attribute «(» AttributeDeclaration «)»
[63]	<i>AttributeDeclaration</i> ::= AttributeName
[64]	<i>ElementTest</i> ::= element «(» (ElementNameOrWildcard (, TypeName «?»?)?)? «)»
[65]	<i>ElementNameOrWildcard</i> ::= ElementName «*»
[66]	<i>SchemaElementTest</i> ::= schema-element «(» ElementDeclaration «)»
[67]	<i>ElementDeclaration</i> ::= ElementName
[68]	<i>AttributeName</i> ::= QName
[69]	<i>ElementName</i> ::= QName
[70]	<i>TypeName</i> ::= QName

4.2. XPath functions

XPath 2.0 expressions can use more than 200 predefined functions described in [33]. It is also possible for a user to define her own functions with an XSLT `function` template described in Section 5.1. Table 4.1 presents a personal selection of what we consider to be the most useful XPath functions separated in broad categories. The XSL stylesheets of the next chapter will show many more uses of these functions.

Table 4.1. A selection of XPath functions

Document information

`document-uri(node)` a string describing the URI of node

Mathematics

`abs(numeric)` absolute value of numeric
`ceiling(numeric)` smallest integer greater than numeric
`floor(numeric)` biggest integer smaller than numeric

Strings

<code>concat(arg₁, arg₂ ...)</code>	string concatenating the string values of arg _i
<code>string-join(strings, sep)</code>	string concatenating the members of the sequence strings using the string sep as a separator
<code>substring(string, start, length)</code>	string portion with string, from position start (the first character is numbered 1) for length characters; if length is not given, then the rest of the string is returned.
<code>string-length(string)</code>	number of characters in string
<code>normalize-space(string)</code>	string after removing leading and trailing blanks and replacing sequences of one or more than one whitespace character with a single space (#x20)
<code>translate(string, mapStr, transStr)</code>	string in which every character appearing at position N in the mapStr is replaced by the character at position N in the transStr
<code>lower-case</code> <code>upper-case</code> (string)	all lower or upper case version of string
<code>contains(string1, string2)</code>	checks if string1 contains string2
<code>starts-with(string1, string2)</code>	checks if string1 starts with string2
<code>ends-with(string1, string2)</code>	checks if string1 ends with string2
<code>substring-before(string1, string2)</code>	string of chars of string1 before the first occurrence of string2
<code>substring-after(string1, string2)</code>	string of chars of string1 after the first occurrence of string2
<code>matches(string, pattern)</code>	checks if string matches the regular expression pattern; unless ^ and \$ are used as anchors, the string matches if one of its substring does
<code>replace(string, pattern, replacement)</code>	string obtained after replacing within string every non-overlapping occurrence of pattern by replacement
<code>tokenize(string, pattern)</code>	break string into a sequence of strings, treating every substring matching pattern as a separator

Sequences

<code>distinct-values(seq)</code>	sequence of different values that appear at least once in seq
<code>remove(seq, position)</code>	sequence of all elements of seq except for the one at position
<code>reverse(seq)</code>	sequence of all elements of seq but in reverse order
<code>subsequence(seq, start, length)</code>	sequence of elements of seq, from position start (the first element is numbered 1) for length elements; if length is not given, then the rest of the sequence is returned.
<code>count(seq)</code>	number of elements within seq
<code>avg</code> <code>max</code> <code>min</code> <code>sum</code> (seq)	average maximum minimum sum of the values within seq

Context

<code>position()</code>	position of the context item within the current sequence of items
<code>last()</code>	the size of the current sequence of items
<code>current-date</code> <code>current-time</code> ()	date or time value of now

A selection of XPath functions (taken from [6]) grouped in categories. The names of the formal parameters are chosen to indicate the expected type or use of the actual parameter. When many functions with the same parameters have a similar behavior, their names are separated by a vertical bar (|); the actual call uses only one choice.

4.3. XPath examples

Example 4.1. XPath expression examples applied on Example D.1 (page 243)

```

/cellar-book/owner ❶
/cellar-book/cellar/wine[quantity<2] ❷
/cellar-book/cellar/wine[1] ❸
//postal-code/.. ❹
/cellar-book/owner/street ❺
//wine/@code ❻
//cat:wine/@codea ❼
//wine[last()]/@code ❽
/cellar-book/cellar/wine[1]/comment/cat:bold ❾
sum(/cellar-book/cellar/wine/quantity) ❿
for $w in //wine return ⓫
    concat($w/quantity, ':', //cat:wine/@code[.=$w/@code]/../@name)

//cat:wine[cat:origin/cat:country='France' and cat:price<20] ⓬

```

- ❶ The owner element of the cellar. Result : node at line 103.
- ❷ The wines for which we have 2 bottles or less. The nodes returned are the wine elements even though the predicate uses `quantity`, an internal element; the predicate is evaluated in the current context of the path specified. When XPath expressions are used in the context of an XSL file, as it is most often the case, the `<` must be replaced by `<` (even within strings!). Result : nodes at lines 131 and 136.
- ❸ The first wine of the cellar. Result: node at line 120.
- ❹ The elements which contain a `postal-code` element. This is achieved by finding a `postal-code` anywhere in the tree from the root and then getting the parent element. Result: nodes at lines 103 and 113.
- ❺ The street of the cellar's owner. Result: 1234 rue des Châteaux
- ❻ The value of the `code` attribute for all wines in the cellar. Result: C00043125, C00312363, C10263859, C00929026.
- ❼ The value of the `code` attribute for all wines in the catalog (note the use of the namespace prefix). Result: C00043125, C00042101, C10263859, C00312363, C00929026.
- ❽ The code of the last wine of the cellar. Result: C00929026.
- ❾ The `cat:bold` element (note again the use of the namespace prefix) within the comment of the first wine of the cellar. Returns `Guy Lapalme, Montréal`(expanded from the entity `&GL;`)
- ❿ Total number of bottles in the cellar by applying the predefined XPath function `sum` to the value of all `quantity` elements of the wines in the cellar. Returns 14.
- ⓫ Sequence of 4 strings each giving the number of bottles of each wine in the cellar, followed by a colon and the name of the corresponding wine: 2:Domaine de l'Île Margaux, 5:Mumm Cordon Rouge, 6:Château Montguéret, 1:Prado Rey Roble.
- ⓬ Sequence of French wines in the catalog costing less than 20 dollars: wines that start on lines 24 and 42.

4.4. Additional Information on XPath

The primary source of information on XPath is the reference [6] but other web sites and books are also interesting

<http://www.w3.org/Style/XSL/> is the best starting point to get information on XSL (including both XPath 1.0 and XPath 2.0) with links to tools and tutorials.

XSLT 2.0 and XPath 2.0 Programmer's reference (4e ed.) [27] A complete and didactic description of all aspects of XPath and XSLT.

Chapter 5. Document Transformation

Since XML is a tree-structured representation of information, it should be relatively simple to change its shape or select parts provided we have a way of to identify subtrees and to combine them in new trees. To achieve this, XML designers have defined the *eXtensible Stylesheet Language* (XSL) technology [28] which refers to two components:

XSLT a transformation language for converting an XML document into either another XML document, into HTML, or into a plain text document (a very wide one-level tree!).

XSL-FO a platform- and media-independent formatting language composed of a set of XML elements, called formatting objects, that describe parts of a printed page at a high level, e.g. `<block>`, `<table>`, etc. These elements are most often produced by XSLT transformations of an XML document.

XSLT is an XML based formalism for defining production rules (similar to OPS5 or Prolog without unification) that match nodes in a tree of an XML document and produce a new tree. These rules are defined in *stylesheets* (XML files named with the `.xsl` extension) that can be validated with a predefined XSLT Schema. This transformation mechanism is very general and can be used to produce any kind of tree, but most often it is used for presentation, one simple kind of tree being an HTML document. In fact, most web browsers can process XML documents linked with XSLT stylesheets to display the resulting transformation. When no stylesheet is associated with the XML file, most browsers now have a predefined stylesheet to *explore* them gradually by expanding and collapsing elements. XSLT depends on XPath [17], explained in the previous chapter, a sublanguage designed to identify nodes in an XML document.

Unfortunately, browsers only implement XSLT 1.0 which is limited compared to the XSLT 2.0 recommendation used in this document. Anyway experience has shown that it is not very reliable to rely on the XSLT processing performed by browser because each one has its peculiarities. Usually the transformation from XML to HTML will be performed on the server. In 2017, the XSLT 3.0 recommendation was approved featuring streaming and modularization, but as these features are not needed in this simple tutorial, we use only XSLT 2.0.

We will see in Section 5.2.1 how to create an HTML tabular presentation of our wine catalog. Section 5.2.2 illustrates features of stylesheets that allow to better select information and perform some simple calculations to produce information that was not present in the original XML file. In Section 5.2.3, we will show how to transform our cellar book instance document into an HTML page with indented bulleted lists. We will then show, in Section 5.3, how to transform our XML instance document into the compact text representation we presented in Figure 1.4. We will illustrate in Section 5.4 the use of *Formatting Objects* to produce a PDF output from an XML document. Finally, in Section 5.5 we compare the Cascading Style Sheets most often associated with HTML pages with XSLT.

The next chapter will describe how to achieve these transformations using XQuery.

The PDF version of this document and the corresponding set of HTML files on the companion web site were produced by XSLT stylesheets from a set of XML source files. This process is explained in Appendix C.

5.1. XSL Transformations

A template to transform a tree node has the following form:

```
<xsl:template match="pattern expr">
  value replacing the matching node(s)
</xsl:template>
```

Like with XML Schemas, we must distinguish between the XSLT predefined elements and the elements used to create the document. The namespace `xsl` is most often used for XSLT elements. In order to trigger a `xsl:template` (a production rule), the transformation process must first identify the node (or nodes) to which it applies. This is done with the value of the `match` attribute that specifies an *pattern expression*, a somewhat restricted type of XPath expression. The content of the `xsl:template` defines the structure of the produced tree by combining any part of the matched tree, new parts or even other parts of the source document tree. The parts of the tree used as building blocks are referenced by XPath expressions and combined with functions, conditions, restricted looping constructs, etc. But the reader must remember that XSLT is a declarative language (similar to Prolog in some ways), so the ordering of templates cannot be reliably used to influence the order of processing of the document tree.

A stylesheet follows a simple process: find a node for which a template applies and then, according to the content of the template, build a new tree structure in the *context* of this node. A context gives access to the current node, its parent, its siblings and its position within its siblings. To build the new tree structure, a template usually involves the application of templates to children of the current node and their combination. This is done with the `xsl:apply-templates` element; without attributes, this forces the application of templates to all the children element nodes of the current node, but the transformation can be applied to other nodes by using the `select` attribute which specifies an XPath expression.

Templates can also be named and called with `xsl:call-template`, similar to procedures in standard programming languages. But be aware that these procedures are *rules* and that they cannot have variables that can change their value. XSLT is thus a single assignment language much like functional languages; parameters are the only mean of passing variable information between templates. Contrarily to ordinary templates, named templates do not change the context of their application. Therefore, we see that the principles underlying XSLT are general, simple and powerful.

Table 5.1 shows the main stylesheet elements for defining transformation rules. As a stylesheet is itself an XML document, it can be validated with the appropriate schema. The root element is `xsl:stylesheet`, which contains a certain number of templates.

`xsl:template` with a `match` attribute will be called when an element matching its pattern is encountered during the processing of the XML instance document. A `xsl:template` element with a `name` attribute must be called explicitly by a `xsl:call-template`. The content of the matched element in the source document is replaced with the content of the template, which usually involves the application of templates to the children of the current node. Formal parameters are declared at the start of an `xsl:template` element with `xsl:param` elements.

A `xsl:function` element with a `name` attribute and `xsl:param` elements can be used to define an XPath function. User-defined function must be declared in a separate namespace in order to differentiate them from the predefined system functions. Even though, user-defined function formal parameters are declared with a `xsl:param` elements, the actual parameters are given within parenthesis in the order of the declar-

ation of the `xsl:param` elements. Of course, the name of the user-defined function call must be preceded by the namespace prefix corresponding to the namespace used in the declaration.

`xsl:apply-templates` is the fundamental operation for traversing the document tree. Without attributes, it indicates that processing should be recursively applied to all its children elements and text nodes (not to its attributes). If the element is empty, then the nodes are processed in the document order but it is possible to specify a different ordering with a `xsl:sort` element. Actual parameters can also be given by name and value using `with-param` elements.

`xsl:value-of` is the fundamental way of getting the *string* value contained in an element from the source document. `xsl:copy-of` should be used to get the whole tree value.

A local *single assignment* variable can be defined within templates with an `xsl:variable` element. Its *string* value can then be recovered in an XPath expression by prefixing its name with `$`. If the whole tree value is needed we can use `xsl:copy` to get a reference to the original value and `xsl:copy-of` to get a new copy.

Conditional processing can be achieved with `xsl:if` which returns its content when the value of its `test` attribute is `true` or a non-empty set of nodes. Also `xsl:choose` can select the first value from a series of alternatives indicated by `xsl:when`. The `xsl:when` conditions are tested in sequence and the first one that returns `true` is the value of this element. If no test succeeds and an `xsl:otherwise` element is present, its value is the value of the `xsl:choose` element.

Although recursive traversal of the document tree is the preferred way of going through nodes, it is also possible to do this traversal iteratively with a `xsl:for-each` and `xsl:for-each-group`. These templates are especially convenient for transforming a sequence of children nodes at all the same levels, for example to create tables or summary information.

Node processing is usually conducted in document order but the order can be changed using `xsl:sort`, which allows to specify the sorting key with the `select` attribute and an ascending or descending sort according to the value of `order` attribute. The elements are usually sorted by their text value but their string value can also be used when sorting by specifying a `data-type` attribute.

In the XPath 1.0, the expression language was quite limited so all control structures (function definition and calls, alternative or loop constructs) for the computation of values were the ones of XSLT. But as we have seen in the previous chapter, with XPath 2.0 we now have the choice between `xsl:if`, `xsl:choose` and `xsl:for-each` and the `if () then ... else` and `for` constructs of XPath. User-defined functions of XPath 2.0 or XSLT named templates can also be sometimes used interchangeably.

In most cases, this choice is a matter of personal choice but if the function has to be used in other XPath expressions then it should be written as a function and not as a named template. Within an XPath function, then the `if` and `for` XPath 2.0 must be used. If the function or expression takes atomic values (string or a number) as input and returns an atomic value, then using an XPath 2.0 expression or function is more convenient than using a template. Named templates are most useful for cases when there are default values, when named parameters are useful or when the current context is needed. In the latter case, the current node would have to be passed as parameter to an XPath function because an XPath function does not execute in a given context. In practice with XSLT 2.0, templates are almost exclusively used for producing results that will appear in the output document. Functions and expressions are more convenient when their result is used by other parts of the program.

Although XPath expressions can be used to reference any node or sequence of nodes in an XML file, it is often convenient and much more efficient to have a direct access to any node in a document. The `xsl:key` element is a top-level declaration indicating to the XSLT processor that a direct access is needed for nodes matching a given pattern. This declaration usually implies the building of an index or hash table that allows a direct matching between a key and a set of nodes matching it. A single key can refer to a many nodes and a single node can be referred by many keys. Access to the nodes is achieved with the XPath key function to which the key value is given. As there can be many key declarations in a program, the key function must also be given in which index the search is to be made.

The dynamic creation of target document elements, attributes and text nodes is performed using the `xsl:element`, `xsl:attribute` and `xsl:text` elements. `xsl:attribute-set` is useful for grouping many attributes under a single name and to combine them.

`xsl:message` is a very convenient tracing device which writes its content either on the standard output or another device. It has no effect on the resulting value.

We will now look at examples of these principles and XSL elements in the following sections. First with straightforward transformations into HTML, then into plain text and finally into *formatting objects* to produce more complex formatting.

Table 5.1. XSLT syntax reminder

<pre><xsl:stylesheet> xsl:import*, (declaration/xsl:variable/xsl:param)* </xsl:stylesheet></pre>
<pre><xsl:template match="pattern" name="QName"> xsl:param*, sequence-constructor* </xsl:template></pre>
<pre><xsl:param name="QName" select="expression"> sequence-constructor </xsl:param></pre>
<pre><xsl:apply-templates select="expression"> (xsl:sort*/xsl:with-param)* </xsl:apply-templates></pre>
<pre><xsl:call-template name="Qname"/></pre>
<pre><xsl:with-param name="QName" select="expression"> sequence-constructor </xsl:with-param></pre>
<pre><xsl:function name="QName"> xsl:param*, sequence-constructor* </xsl:function></pre>
<pre><xsl:value-of select="expression"> sequence-constructor </xsl:value-of></pre>
<pre><xsl:variable name="QName" select="expression"> sequence-constructor </xsl:variable></pre>
<pre><xsl:copy> sequence-constructor </xsl:copy></pre>
<pre><xsl:copy-of select="expression"/></pre>

<pre><xsl:if test="expression"> <i>sequence-constructor</i> </xsl:if></pre>
<pre><choose> <i>xsl:when*</i>, <i>xsl:otherwise?</i> </choose></pre>
<pre><xsl:when test="expression"> <i>sequence-constructor</i> </xsl:when></pre>
<pre><xsl:otherwise> <i>sequence-constructor</i> </xsl:otherwise></pre>
<pre><xsl:for-each select="expression"> <i>xsl:sort*</i>, <i>sequence-constructor</i> </xsl:for-each></pre>
<pre><xsl:for-each-group select="expression" group-by="expression"> <i>xsl:sort*</i>, <i>sequence-constructor</i> </xsl:for-each-group></pre>
<pre><xsl:sort select="expression" data-type="{string}"> <i>sequence-constructor</i> </xsl:sort></pre>
<pre><xsl:key name="qname" match="pattern" use="expression"> <i>sequence-constructor</i> </xsl:key></pre>
<pre><xsl:element name="{string}"> <i>sequence-constructor</i> </xsl:element></pre>
<pre><xsl:text> <i>character data</i> </xsl:text></pre>
<pre><xsl:attribute name="{string}" select="expression"> <i>sequence-constructor</i> </xsl:attribute></pre>
<pre><xsl:attribute-set name="QName" use-attribute-sets="Qnames"> <i>xsl:attribute*</i> </xsl:attribute-set></pre>
<pre><xsl:message> <i>sequence-constructor</i> </xsl:message></pre>

A reminder of the subset of the XSLT syntax used in our examples. Names in italics refer to other elements. Regular expression syntax is explained in Table B.1

5.2. Transformation in HTML

5.2.1. Table

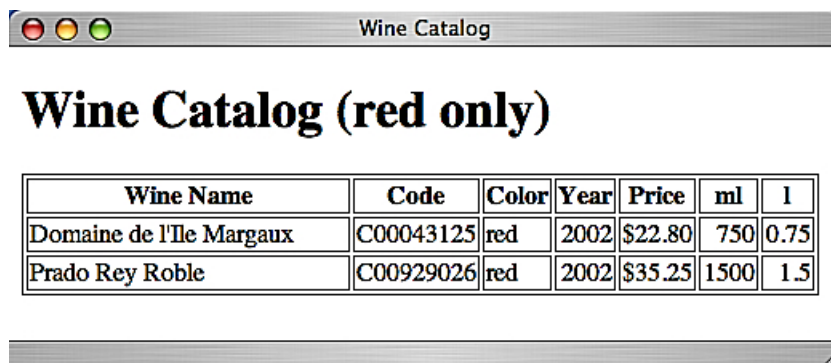
XSLT was designed from the start to transform trees into other trees. One type of tree that is easy to produce with XSLT is a well-formed HTML document, most often an XHTML document, which is a valid XML

file that can also be displayed directly by web browsers. In this section, we will first show a simple selection of information from an XML file to produce tabulated information displayed as an XHTML page.

When the body of a template contains XML elements that are not XSLT elements (i.e. transformation instructions), they are copied verbatim to the output. So it is relatively easy to build the structure of an HTML document in which only some parts will be processed. This is similar in principle to the *backquote macro* processing in Lisp.

Selecting only red wines in our wine catalog (Example 2.3 (page 12)) and outputting an HTML table of a subset of the available information for each (Figure 5.1) can be done with the XSLT stylesheet given in Example 5.2.

Figure 5.1. Web browser rendering of Example 5.1 produced by running Example 5.2 on Example 2.3



The screenshot shows a web browser window with the title "Wine Catalog". The main content is a heading "Wine Catalog (red only)" followed by a table with the following data:

Wine Name	Code	Color	Year	Price	ml	l
Domaine de l'Ile Margaux	C00043125	red	2002	\$22.80	750	0.75
Prado Rey Roble	C00929026	red	2002	\$35.25	1500	1.5

Example 5.1. HTML tabular output of the red wines of the catalog produced by Example 5.2

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
>
<html xmlns:cat="http://www.iro.umontreal.ca/lapalme/wine-catalog"
  xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Wine Catalog</title>
  </head>
  <body>
    <h1>
      Wine Catalog (red only)
    </h1>
    <table border="1">
      <tr>
        <th>Wine Name</th>
        <th>Code</th>
        <th>Color</th>
        <th>Year</th>
        <th>Price</th>
        <th>ml</th>
        <th>l</th>

```

```

</tr>
<tr>
  <td>Domaine de l'Île Margaux</td>
  <td>C00043125</td>
  <td>red</td>
  <td align="right">2002</td>
  <td align="right">$22.80</td>
  <td align="right">750</td>
  <td align="right">0.75</td>
</tr>
<tr>
  <td>Prado Rey Roble</td>
  <td>C00929026</td>
  <td>red</td>
  <td align="right">2002</td>
  <td align="right">$35.25</td>
  <td align="right">1500</td>
  <td align="right">1.5</td>
</tr>
</table>
</body>
</html>

```

The stylesheet in Example 5.2 first defines a template matching the root node (line 14-❶). Because the wine catalog is defined in a specific namespace, its prefix must be declared (line 3-❶) and used for selection. This template outputs the overall structure of the XHTML file and then calls `xsl:apply-templates` to search for an appropriate template on each child. In this case, it will correspond to `cat:wine-catalog`. The corresponding template (line 23-❷) outputs a global heading and then starts a table and defines its headers; the lines of the table will be filled by selecting (line 35-❸) wines whose color property is red. To set up this color filter, a value must be assigned to the `code` variable either with an `xsl:variable` or `xsl:parameter` element. Here we chose the latter (line 6-❷). Since color is defined as a parameter of the stylesheet, it is possible to change its value when the stylesheet is called by the XSLT processor. The way of setting this value from outside the stylesheet depends on each processor. In order for the `select` attribute value to be the character string 'red' and not the value associated with the node red (which is empty at this moment), single quotes must be added within the double quotes indicating the value of the attribute. Note again the use of the namespace prefix. The predicate used in square brackets limits the nodes to which the templates will be applied but does not change the context of the node, which is still the wine node.

The output for each selected wine is defined with a template that is applied to each `cat:wine`. It outputs, on a single row of the table, the values of its attributes, its color, its year (right aligned) and formats its price to start with a dollar sign (right aligned). It finally outputs the volume of each bottle in milliliters and in liters. Because the information in the wine catalog is not given in milliliters, we call both a user-defined XPath function (line 49-❹) and a *named* template (line 52-❺) to transform it appropriately. We have deliberately chosen the same program structure in both the function and the named template to highlight the differences and the similarities between these two means of organizing computation in a stylesheet. Usually we will use functions to compute strings and templates to produce elements, but the choice is left to the programmer.

Example 5.2. [wineCatalog.xsl] XSLT stylesheet designed to select the red wines in the catalog (Example 2.3) and to produce Example 5.1 displayed as Figure 5.1

```

1 <?xml version="1.0" encoding="UTF-8" ?>
  <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:cat="http://www.iro.umontreal.ca/lapalme/wine-catalog"
    xmlns="http://www.w3.org/1999/xhtml"
5    version="2.0">
  <xsl:param name="color" select="'red'"/>
  <!-- to produce legal and validable XHTML ... -->
  <xsl:output method = "xml"
10    doctype-public = "-//W3C//DTD XHTML 1.0 Strict//EN"
    doctype-system =
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
    indent = "yes" encoding = "UTF-8"/>

  <xsl:template match="/">
15    <html>
      <head><title>Wine Catalog</title></head>
      <body>
        <xsl:apply-templates/>
      </body>
20    </html>
  </xsl:template>

  <xsl:template match="cat:wine-catalog">
25    <h1>
      Wine Catalog (<xsl:value-of select="$color"/> only)
    </h1>
    <table border="1">
      <tr>
30        <xsl:for-each
          select="'Wine Name', 'Code', 'Color', 'Year', 'Price', 'ml', 'l'">
          <th><xsl:value-of select="."/></th>
        </xsl:for-each>
      </tr>
      <xsl:apply-templates
35        select="cat:wine[cat:properties/cat:color=$color]"/>
    </table>
  </xsl:template>

  <xsl:template match="cat:wine">
40    <tr>
      <td><xsl:value-of select="@name"/></td>
      <td><xsl:value-of select="@code"/></td>
      <td><xsl:value-of select="cat:properties/cat:color"/></td>
      <td align="right"><xsl:value-of select="cat:year"/></td>
45    <td align="right">
        <xsl:value-of select="format-number(cat:price, '$0.00')"/>
      </td>
      <td align="right">
        <xsl:value-of select="cat:toML(@format)"/>
50    </td>
    </tr>
  </xsl:template>

```

```

50         </td>
           <td align="right">
             <xsl:call-template name="toL">
               <xsl:with-param name="fmt" select="@format"/>
             </xsl:call-template>
55         </td>
       </tr>
     </xsl:template>

     <xsl:function name="cat:toML">
60       <xsl:param name="fmt"/>
       <xsl:value-of select="
         if ($fmt='375ml') then '375'
         else if ($fmt='750ml') then '750'
65         else if ($fmt='1l') then '1000'
         else if ($fmt='magnum') then '1500'
         else 'big'"/>
     </xsl:function>

     <xsl:template name="toL">
70       <xsl:param name="fmt"/>
       <xsl:choose>
         <xsl:when test="$fmt='375ml'">0.375</xsl:when>
         <xsl:when test="$fmt='750ml'">0.75</xsl:when>
         <xsl:when test="$fmt='1l'">1.0</xsl:when>
75         <xsl:when test="$fmt='magnum'">1.5</xsl:when>
         <xsl:otherwise>big</xsl:otherwise>
       </xsl:choose>
     </xsl:template>
   </xsl:stylesheet>
80

```

- ❶ Definition of the `cat` namespace prefix in order to access the elements of the wine catalog which are defined in this namespace. The default namespace is set to be the one needed for a valid XHTML file. With this declaration HTML tags that are used in the stylesheet are in the appropriate namespace for HTML validation.
- ❷ Global parameter initialized with a string value; note the use of the internal single quotes to make sure that the string value is used and not a reference to a (non-existing) `red` element. This value can be overridden by a run-time parameter when the stylesheet is applied.
- ❸ `xsl:output` declaration to tell the transformation engine to serialize with the appropriate headers to produce valid XHTML.
- ❹ Template matching the root node that defines the skeleton of the HTML page: a head and a body with a call to apply templates on its children elements (only `cat:wine-catalog` in this case).
- ❺ Template matching the catalog node to produce a header with a title and a table. The headings of the table are defined as the first row. The rest of the table will be filled with lines produced by each wine.
- ❻ Uses a `for-each` in a sequence of strings to output the headers in the first row of the table.
- ❼ Selects the wines with the chosen `color`. The result is a sequence wine nodes. The template on line 39-❽ is applied to each of them.
- ❽ Outputs properties of a wine. The first four are output as they appear in the source. The `price` is formatted in dollars and cents and the bottle format is given in either milliliters or liters.

- ⑨ Outputs the bottle capacity, expressed in milliliters, using a user-defined XPath function (line 59-⑩). To distinguish user-defined functions from system defined ones, user-defined functions must be declared in a separate namespace. Here we simply chose the `cat` namespace but we could have used a different one.
- ⑩ Calls a *named* template that outputs the bottle capacity, expressed in liters.
- ⑪ A user defined function which must be defined in a specific namespace and that can be called within an XPath expression like any other system defined one. Here we simply use a multi-line XPath expression to select the appropriate case within a cascaded `if (...) then ... else ...` expression. Note the definition of an `xsl:param` element to define the formal parameter that will be used as the value of the actual parameter when the function is used in an XPath expression.
- ⑫ Definition of a named template in order to output the number of liters corresponding to the value of the `fmt` formal parameter. The formal parameter is referred to by the XPath expression `$fmt` within the template. The template chooses the value to return depending on the value of the `format` attribute of the current node given as actual parameter when the named template is called (line 52-⑬).

5.2.2. Computing New Information

XSLT can also be used for more complex selections and transformations. We will now show how to create a web page presenting the content of the cellar and integrating information from the wine catalog. The end result is shown in Figure 5.2 (an outline of the underlying HTML code is shown in Example 5.3). There are external links in order to get more information about the wines by googling for the name of wine. There are two similar links for each wine but it is just to show and compare ways of creating them in XSL.

Figure 5.2. HTML rendering of Example 5.3

Cellar of Jude Raisin

Personal address	Cellar address
1234 rue des Châteaux St-George ON M7W 7S0	4587 des Futailles Vallée des crus QC H3C 4J8

Code	Name	Purchase Date	Rating	Nb bottles
00043125	Domaine de l'Île Margaux <i>Domaine de l'Île Margaux</i>	2005-06-20		2
00312363	Mumm Cordon Rouge <i>Mumm Cordon Rouge</i>	2004-11-19	***	5
00871996	Château Montguéret <i>Château Montguéret</i>	2005-06-19		0
00929026	Prado Rey Roble <i>Prado Rey Roble</i>	2003-10-15		1
Estimated value			245.85	8

Comments

C00043125 : **Guy Lapalme, Montréal:** should reorder soon
 C00312363 : Bottle too small...
 C00871996 : Really great
 C00929026 : for **big** parties

Example 5.3. HTML output by the XSLT code is shown in Example 5.4

This HTML code has been slightly reformatted and trimmed to fit on the page.

```
<html>
  <head>
    <title>Cellar of Jude Raisin</title>
  </head>
  <body>
    <h1>Cellar of Jude Raisin</h1>
    <table border="1">
      <tr><th>...</th><th>...</th></tr>
      <tr>
        <td>...</td>
        <td>...</td>
      </tr>
    </table>
    <p/>
    <table border="1">
```

```

<tr><th>...</th><th>...</th><th>...</th>
  <th>...</th><th>...</th>
</tr>
<tr>
  <td>...</td>
  <td>...</td>
  <td align="right">...</td>
  <td align="center"/>
  <td align="right">...</td>
</tr>
...
<tr>
  <td colspan="3">...</td>
  <td align="right">...</td>
  <td align="right">...</td>
</tr>
</table>
<h3>Comments</h3>C00043125 :<b>Guy Lapalme, Montréal</b>: should reorder soon<br/>
C00312363 : Bottle too small...<br/>
C00929026 :for <b>big</b> parties<br/>
C10263859 :Really great<br/>
</body>
</html>

```

Example 5.4 illustrates some new features. The root template (line 20-❶) creates the high-level structure of the XHTML file: the title of the page, also displayed at the top of the page, refers to the name of the owner. In Example 2.2, element name (line 12-❷) is structured in two elements: `first` and `family`. When the value of such an element is returned from an `xsl:value-of`, it is the text content of all elements. In fact, there are 5 parts in this case:

- the text node comprising a `\n` (this is the notation for an *end of line character*) following the name opening tag and white space until the start of `first`
- the content of the element `first`
- the `\n` and spaces between the closing tag of `first` and the opening tag of `family`
- the content of the element `family`
- the `\n` and spaces between the closing tag of `family` and the closing tag of `name`

Given the fact that a sequence of `\n` and spaces in HTML is displayed as a single space by the browser, we get an appropriate display in this case. But this shows that handling text content can become a bit tricky. The next section will explain how to work with some of the most frequent cases.

The content of the cellar book is obtained by an implicit call to the template defined on line 36-❸ which creates a table with the address of the owner (line 43-❹) and the cellar (line 47-❺). It then calls the `cellar` template (line 54-❻). The lines of the addresses are obtained by looping on all elements with a `for-each` and outputting a `
` between the text values of each element of the `owner` and `location` elements. Because we want to skip the first element (the owner name has already been given at the top), we only keep elements (line 43-❹) with a position number greater than 1.

The `cellar` (line 58-❼) template produces a table of information about wines in the cellar, sorted by their code. This is why the content of the `xsl:apply-templates` element (line 66-❼) is an `xsl:sort`

element indicating the sorting key and the sort order. The last line of the table contains an estimated total value of the cellar obtained by a call (line 73-⑩) to the `total` named template. To compute the total number of bottles (line 79-⑪), we can use the predefined `sum` function. Finally, if there are any comment in the wine elements of the cellar (line 84-⑫), we add a `Comments` section and output each of them, also in increasing order of wine code so that they are in the same order as in the table of wines.

We define the `wine` template (line 95-⑬) to create a row in the table of wines. In order to gather some information about this wine from the wine catalog, we pass the wine node from the wine catalog as a parameter to the call to the `nameAndUrl` template (line 99-⑭). The link between the current element and the corresponding element in the catalog is made using the value of the `code` variable given in the XPath expression. The remaining elements of the row are the purchase date (right-aligned), a number of stars corresponding to the rating and the quantity (right-aligned). The estimated value of the cellar is computed using XPath expressions.

`nameAndUrl` is a named template that receives a wine element as parameter. From this wine element, we create a XHTML link (line 117-⑰) with an `a` element with an `href` attribute whose value is a string specifying the request to send to Google to search for this wine using its name. This is a bit involved because the link must be created dynamically using the `xsl:attribute` elements added to an enclosing `a` element with appropriate contents. For that, we use two variables:

`$GoogleStart` set in `xsl:param` for the whole stylesheet (line 8-①). This string corresponds to the CGI call to Google to search for the name of the wine.

`$name` is the name of the wine found in the catalog.

These values are used to create the value of the attribute `href` of the `a` element in the resulting HTML code.

Because creating such dynamic elements and attributes is often required, XSLT designers have defined a *non-XML* formalism called *Attribute Value Template* in which we put braces around XPath expressions that denote values that will be evaluated at run time and whose string value will replace the braces and their content (line 127-⑱). One should consult the XSLT documentation to determine in which contexts this shortcut notation is allowed (it is indicated by surrounding the content of the quotes by braces).

The `comment` template (line 134-⑲) outputs the value of the `code` of the parent node, a colon and the text value of the comment followed by a line break. In getting the content of the comment, the `cat:bold` elements will be processed by the appropriate template (line 142-⑳) that will transform them in HTML `b` tags.

Example 5.4. [CellarBook.xsl] XSLT stylesheet to produce information about the cellar (Example 2.2). The resulting HTML code (Example 5.3) is rendered as Figure 5.2.

```

1 <?xml version="1.0" encoding="UTF-8"?>
  <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:cat="http://www.iro.umontreal.ca/lapalme/wine-catalog"
    xmlns="http://www.w3.org/1999/xhtml"
5    version="2.0">

    <!-- part of URL for a Google search -->
    <xsl:param name="GoogleStart">http://www.google.com/search?q=</xsl:param>①

```

```
10  <!-- to produce legal and validable XHTML ... -->
    <xsl:output method="xml"
      doctype-public="-//W3C//DTD XHTML 1.0 Strict//EN"
      doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
      indent="yes"
15  encoding="UTF-8"/>

    <xsl:key name="catalog" match="cat:wine" use="@code"/> ❷

    <!-- matches the root node -->
20  <xsl:template match="/"> ❸
      <html>
        <head>
          <title>Cellar of <xsl:value-of
            select="cellar-book/owner/name"/>
25          </title>
          </head>
          <body>
            <xsl:apply-templates/>
          </body>
30  </html>
    </xsl:template>

    <!-- output of the content of the cellar:
      addresses of the owner and cellar
35  followed by a table of wines -->
    <xsl:template match="cellar-book"> ❹
      <h1>Cellar of <xsl:value-of select="owner/name"/></h1>
      <table border="1">
        <tr>
40          <th>Personal address</th>
          <th>Cellar address</th>
        </tr>
        <tr><td><xsl:for-each select="owner/*[position()>1]"> ❺
          <xsl:apply-templates/><br/>
45          </xsl:for-each>
          </td>
          <td><xsl:for-each select="location/*"> ❻
          <xsl:apply-templates/><br/>
          </xsl:for-each>
50          </td>
        </tr>
      </table>
      <p/>
      <xsl:apply-templates select="cellar"/> ❼
55  </xsl:template>

    <!-- content of the cellar as a table -->
    <xsl:template match="cellar"> ❽
      <table border="1">
60        <tr><!-- head of the table -->
          <xsl:for-each
            select="'Code', 'Name', 'Purchase Date', 'Rating', 'Nb bottles'">
```

```

        <th><xsl:value-of select="."/></th>
    </xsl:for-each>
65    </tr>
    <xsl:apply-templates select="wine"><!-- each row --> 9
        <xsl:sort select="@code" order="ascending"/>
    </xsl:apply-templates>
    <tr>
70    <td colspan="3">Estimated value</td>
        <td align="right">
            <!-- compute the estimated value of the cellar -->
                <xsl:value-of select=" 10
                    sum (for $w in wine return
75    $w/quantity * key('catalog', $w/@code)/cat:price)"/>
            </td>
            <td align="right">
                <!-- compute the total number of bottles -->
                <xsl:value-of select="sum(wine/quantity)"/> 11
            </td>
80    </tr>
</table>
<!-- put comment section if at least one comment appears -->
<xsl:if test="count(wine/comment)>0"> 12
85    <h3>Comments</h3>
        <p>
            <xsl:apply-templates select="wine/comment">
                <xsl:sort select="../@code" order="ascending"/>
            </xsl:apply-templates>
90    </p>
    </xsl:if>
</xsl:template>

<!-- information about a wine -->
95    <xsl:template match="wine"> 13
        <tr>
            <td><xsl:value-of select="substring(@code,2)"/></td>
            <td>
                <xsl:call-template name="nameAndUrl"> 14
100    <xsl:with-param name="wine" select="key('catalog', @code)"/>
                </xsl:call-template>
            </td>
            <td align="right"><xsl:value-of select="purchaseDate"/></td>
            <td align="center">
105    <xsl:value-of select="substring('*****',1, 15
                if (rating/@stars) then rating/@stars else 0)"/></td>
            <td align="right"><xsl:value-of select="quantity"/></td>
        </tr>
    </xsl:template>
110

<!-- output the name of the wine with a link for "googling" it -->
<xsl:template name="nameAndUrl"> 16
    <xsl:param name="wine"/>
115    <xsl:variable name="name" select="encode-for-uri($wine/@name)"/>
    <!-- dynamic creation of an element and its attributes -->

```

```

120     <a>
        <xsl:attribute name="href">
            <xsl:value-of select="$GoogleStart"/>
            <xsl:value-of select="$name"/>
        </xsl:attribute>
        <xsl:value-of select="$wine/@name"/>
    </a>
125 <!-- link creation using an Attribute Value Template -->
    <br/>
    <i>
        <a href="{ $GoogleStart } { $name }">
            <xsl:value-of select="$wine/@name"/>
        </a>
130    </i>
</xsl:template>

<!-- comment preceded by the code corresponding to it -->
135 <xsl:template match="comment"><xsl:text>
    </xsl:text>
    <xsl:value-of select="../@code"/>
    <xsl:text> : </xsl:text>
    <xsl:apply-templates/><br/>
</xsl:template>

140 <!-- global change of <bold> tags to html <b> tags -->
    <xsl:template match="cat:bold">
        <b><xsl:apply-templates/></b>
    </xsl:template>

145 </xsl:stylesheet>

```

- ❶ Parameter giving the start of the URL allowing a search for a given wine on Google.
- ❷ Declaration of a key to access a wine element of the catalog use its `code` attribute as a key.
- ❸ Template for the root node that defines the skeleton the HTML page: a head with a title indicating the name of the owner and a body to be filled by the application of templates on the child element (`cellar-book` in this case).
- ❹ Global header with the name of the owner and then a table giving more information about the owner and the location of the cellar.
- ❺ Loop over all children nodes of the address, except the first one, the name of the owner. After each node which will be output using the default rules (only their text content will be output), a line break is forced with an HTML `br` element.
- ❻ Loop over all children nodes of the location. After each node which will be output using the default rules (only their text content will be output), a line break is forced with an HTML `br` element.
- ❼ Outputs the content of the cellar with the appropriate template.
- ❽ The content of the cellar is given as a table with a header defined in this template, followed by a series of lines corresponding to each wine and finally a line holding the estimated value of the whole cellar.
- ❾ Applies the template for each wine but in increasing order of the `code` attribute.

- ⑩ The total value of the cellar is computed using a relatively complex XPath expression: that loops over all wines and returns the sum of the value of each wine. The value of a wine is given by the number of bottles (`quantity`) multiplied by the price of the wine in the catalog having the same `code` attribute as this wine. The price in the catalog is found by first accessing the wine element of the catalog with the `key` function giving the value of `code` attribute of the cellar book.
- ⑪ The total number of bottles is the sum of the values of all `quantity` elements.
- ⑫ Comments appear at the bottom of the table if at least one wine has a comment. They are also given in ascending order of `code` attribute.
- ⑬ Outputs a line of an HTML table for a given wine.
- ⑭ Outputs the name of wine and its URL using the named template defined on line 113-⑯. The wine element in the catalog is found using the `key` function.
- ⑮ Outputs a string of `*` corresponding to the number of stars for this wine.
- ⑯ Named template for outputting the name of the wine as the anchor text for an HTML link to Google with the appropriate wine name to get further information about it. We show here two alternative ways of creating the link.
- ⑰ Creates an a HTML element with a computed value for the `href` attribute: this value is the concatenation of the query string and the wine code.
- ⑱ Creates an a HTML element relying on an *Attribute Value Template* in which the content of the braces-enclosed expressions are evaluated as XPath expressions. In many cases, this notation is simpler than the preceding one.
- ⑲ A comment is preceded with the value of the surrounding `code` attribute and followed by a line break.
- ⑳ A `cat:bold` element is transformed into an HTML `b` element.

A observant reader might wonder where the text within a text element is coming from, because there is no template in our program for this case. This output is achieved by means a built-in XSLT template shown in Example 5.5 which stipulates at line 1-① that the content of text nodes and attributes are replaced by their string value. The built-in rule for attribute nodes thus ignores their name.

Another built-in template for the document and element nodes, shown in line 5-②, launches the application of templates on the children nodes thus traversing the document in a depth-first manner (i.e. in document order). The combination of these two built-in rules explains why only the text content of an XML document is displayed in a browser when no specific template is defined.

Example 5.5. [built-in-template-rules.xsl] Built-in template rules for XSLT. These rules are applied when no other defined template can be applied.

```

1 <xsl:template match="text()|@"*">                                ①
    <xsl:value-of select="string(.)"/>
</xsl:template>

5 <xsl:template match="/*">                                        ②
    <xsl:apply-templates/>
</xsl:template>

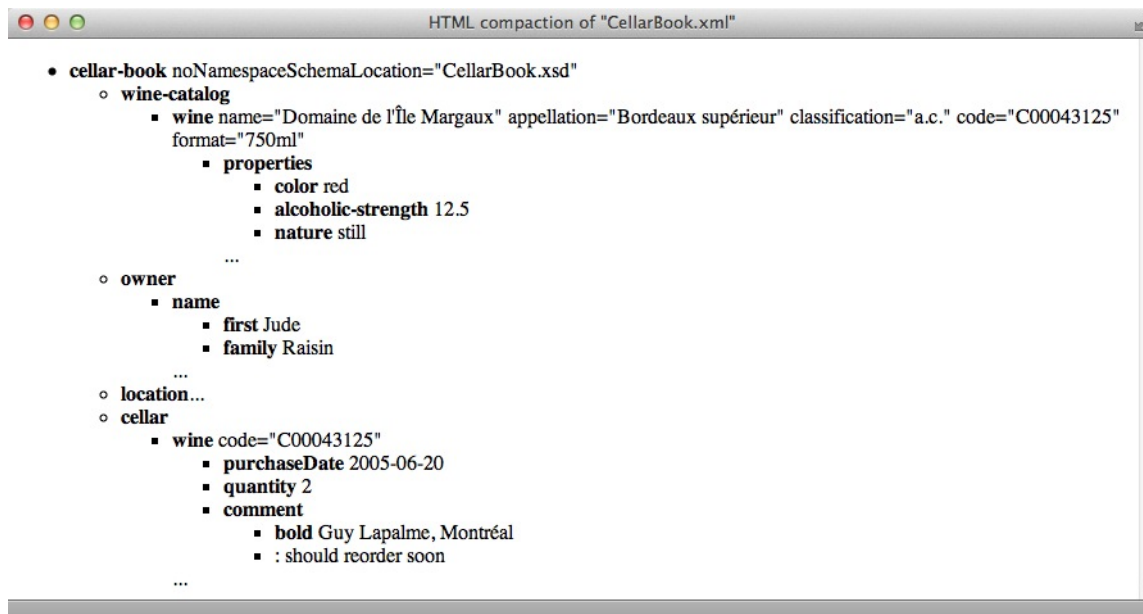
```

- ① A text node or an attribute is replaced by its string content.
- ② A document or element node applies the templates to its children elements.

5.2.3. Bulleted Lists

The previous examples showed transformations specific to a given XML file type. We now describe a transformation that can be applied to any XML file to show its indentation structure by means of nested HTML unnumbered lists.

Figure 5.3. HTML rendering of Example 5.6



Example 5.6. Excerpt of the HTML output (slightly reformatted here to fit in the page) produced by the transformation of Example 5.4 on the cellar book (Example 2.2).

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head><title>HTML compaction of "CellarBook.xml"</title></head>
  <body>
    <ul><li><b>cellar-book</b> noNamespaceSchemaLocation="CellarBook.xsd"<ul>
      <li><b>wine-catalog</b>
        <ul><li><b>wine</b> name="Domaine de l'Île Margaux" appellation="Bordeaux supé
          <ul><li><b>properties</b>
            <ul>
              <li><b>color</b> red</li>
              <li><b>alcoholic-strength</b> 12.5</li>
              <li><b>nature</b> still</li>
            </ul>
          </li>
          ...
        </ul></li>
      </ul></li>
    </ul></li>
  </body>
</html>
```

```

    <li><b>owner</b><ul><li><b>name</b>
      <ul>
        <li><b>first</b> Jude</li>
        <li><b>family</b> Raisin</li>
      </ul>
    </li>
    ...
  </ul>
</li>
<li><b>location</b>...</li>
<li><b>cellar</b>
  <ul><li><b>wine</b> code="C00043125"<ul>
    <li><b>purchaseDate</b> 2005-06-20</li>
    <li><b>quantity</b> 2</li>
    <li><b>comment</b>
      <ul>
        <li><b>bold</b> Guy Lapalme, Montréal</li>
        <li>: should reorder soon</li>
      </ul>
    </li></ul>
  </li></ul>
  ...
</ul>
</li></ul></li></ul>
</body>
</html>

```

To transform the cellar book (Example 2.2) into the HTML code of Example 5.6 (rendered in Figure 5.3) we can use the code given in Example 5.7 which has four templates:

- one matching the root element (line 17-④) that produces the overall structure of the HTML file with its head and body elements. The processing of subelements (line 25-⑥) is called within an unnumbered list delimited by `ul` tags.
- matching attributes (line 31-⑦) is done by outputting a space (with an entity defined on line 3-①), the name of the attribute followed by an equal sign and its value within double quotes.
- elements (line 36-⑧) are transformed by outputting the name of the element returned by the function `local-name()` in bold (line 38-⑨) followed by its attributes. If the element is a node without any children (line 41-⑩) then it is output with only its content, otherwise (line 44-⑪) a new unnumbered list is started and template matching is applied on children nodes (line 46-⑫).
- text node content (line 53-⑬) is output within `li` tags.

The `*` in the `match` attribute (line 36-⑧) indicates that this rule applies to all element nodes not matched by a more specific rule such as the one on line 17-④ that matches only the root node.

Example 5.7. [compactHTML.xsl] XSLT transformation to produce a bulleted outline (Example 5.6) from the cellar book (Example 2.2)

```

1  <?xml version="1.0" encoding="UTF-8"?>
   <!DOCTYPE stylesheet [
   <!ENTITY space "<xsl:text> </xsl:text>">                                ❶
   ]>
5  <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns="http://www.w3.org/1999/xhtml"
   version="2.0">

   <xsl:strip-space elements="*" />                                          ❷
10  <!-- to produce legal and validable XHTML ... -->
   <xsl:output method="xhtml"
   doctype-public="-//W3C//DTD XHTML 1.0 Strict//EN"
   doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
   indent="yes"                                                                ❸
15  encoding="UTF-8" />

   <xsl:template match="/">                                                ❹
   <html>
   <head>
20     <title>HTML compaction of
   "<xsl:value-of select="replace(document-uri(.),'.*/(.*)','$1')"/>"</title>
   </head>
   <body>
   <ul>
25     <xsl:apply-templates/>                                                ❺
   </ul>
   </body>
   </html>
   </xsl:template>

30  <xsl:template match="@*">                                              ❻
   &space;<xsl:value-of select="local-name()"/>
   <xsl:text>=</xsl:text><xsl:value-of select="."/><xsl:text></xsl:text>
   </xsl:template>

35  <xsl:template match="*">                                              ❼
   <li>
   <b><xsl:value-of select="local-name()"/></b>                                ❽
   <xsl:apply-templates select="@*" />
40  <xsl:choose>                                                            ❿
   <xsl:when test="count(*)=0"> <!--single text node ?--> 11
   &space;<xsl:value-of select="."/>
   </xsl:when>
   <xsl:otherwise> <!--possible mixed node--> 12
45  <ul>
   <xsl:apply-templates/> 13
   </ul>
   </xsl:otherwise>
   </xsl:choose>
50  </li>

```



```
    </xsl:template>

    <xsl:template match="text()">
        <li><xsl:value-of select="."/></li>
55    </xsl:template>

</xsl:stylesheet>
```

14

- ❶ Entity that corresponds to an explicit blank space.
- ❷ Ignores blank spaces in all elements.
- ❸ Makes sure that the output will be valid XHTML output.
- ❹ Matches the root node and produce the HTML skeleton.
- ❺ Outputs the title of the HTML page. Because the `document-uri` returns the full uri of the current document, the `replace` function uses a regular expression to keep only the part after the last slash.
- ❻ Start processing the root node of the document by applying the appropriate template.
- ❼ An attribute is output as a space (note the use of an entity to create an explicit text node that will not be ignored afterwards), the name of the attribute and its value between double quotes after an equal sign.
- ❽ An element node is a *list-item*.
- ❾ The name of the element in bold, followed by the attributes using the template defined on line 31-❷.
- ❿ The content of the element is output differently when it has children nodes or not.
- ⓫ There are no children nodes, in our case this will only be a text node, its content is copied in the current list item.
- ⓬ If there are children nodes, a new embedded unnumbered list is started.
- ⓭ The templates are applied on all children nodes, this is the default when no `select` attribute is present.
- ⓮ In the case of a text node within a list of children (i.e. a child of a mixed content element), it is output as list item.

5.3. Transformation into a Compact Textual Form

We will now use an XSLT stylesheet (shown in Example 5.9) to produce the compact form of an XML file. The output of this transformation will only be a stream of *plain* characters without any tags. This shows that XSLT can be used to transform XML input into a plain text file.

Example 5.8. Text compaction of the cellar book of Example 2.2 produced by the stylesheet of Example 5.9.

Some lines and parts of lines indicated by . . . have been omitted here.

```
cellar-book[@noNamespaceSchemaLocation[CellarBook.xsd]
  wine-catalog[wine[@name[Prado Rey Roble]
    @appellation[Ribera-del-duero]
    @classification[d.o.]
    @code[C00929026]
    @format[magnum]
    properties[color[red]
      alcoholic-strength[12.5]
      nature[still]]
    origin[country[Spain]
      region[Old Castille]
      producer[Real Sitio de Ventosilla SA]]
    price[35.25]
    year[2002]]
    wine[...]]
  owner[name[first[Jude]
    family[Raisin]]
    street[1234 rue des Chateaux]
    city[St-George]
    province[ON]
    postal-code[M7W 7S0]]
  location[street[4587 des Futailles]
    city[Vallée des crus]
    province[QC]
    postal-code[H3C 4J8]]
  cellar[wine[@code[C00043125]
    purchaseDate[2005-06-20]
    quantity[2]
    comment[bold[Guy Lapalme, Montréal]
      : should reorder soon]]
    ...
    wine[@code[C00929026]
    purchaseDate[2003-10-15]
    quantity[1]
    comment[for
      bold[big]
      parties]]]]]
```

The algorithm given Example 5.9 follows the same pattern as the one explained in Section 5.2.3. We recursively follow the structure of the tree and output a corresponding stream of characters. In our case, only one rule is applied to all element nodes: we output the name of the element and then output its attributes and children, with an indentation corresponding to the number of characters in the name of the parent element. The root node has an indentation of 0. Because we want an output with few blank lines, we only change line after having written the first attribute or child element. This rule is implemented with the template starting on line 40-⑥. The `*` in the `match` attribute indicates that this rule applies to all element nodes not matched by a more specific rule such as the one on line 8-④ that matches only the root node. In this latter rule, we apply the general rule to all children with `xsl:apply-templates` without any `select` attribute. All templates have a parameter indicating the indentation given as an `xsl:with-param` element and declared in the template with the `xsl:param` element.

Characters are put in the output stream literally by `xsl:text` elements. Text can also be computed with `xsl:value-of` elements whose `select` attribute is an XPath expression. Conditions can be introduced with an `xsl:if` element whose `test` attribute can refer to the current context; here `count(../@*)` counts the number of attributes of the parent and `position()` indicates the rank of the current node among its siblings. With this information, we can decide if the current line should be ended and insert the appropriate number of blanks according to the value of the `indent` parameter before outputting the information on this line. After the output of the element name, we call the templates for all attributes, elements and text nodes of this node (this is achieved with the `select` attribute on line 47-⑨). In our case, we also update the current indentation that will be given to all these nodes. The output of all these recursive template applications will be enclosed in a pair of square brackets.

Because all characters and new lines in the stylesheets are returned as they appear in the input (including `\n` and leading and trailing spaces between elements), it can be difficult to achieve a specific output format. This is not really a problem if the output of the transformation is HTML, because in this case these spurious spaces and newlines are removed before being displayed. In our case, the transformation output is given as is to the user so it is simpler to only output the content of stylesheet elements without any `\n` and without leading and trailing spaces. This is why, on line 5-①, we declare that all elements (`*`) of this stylesheet should ignore all spaces in the instance file.

On line 22-⑤, we define the template to output the value of an attribute, which is simply the name of the attribute preceded by an `@` and followed by its value in square brackets. If the attribute is not the first one, the line is ended and a new indentation is produced.

On line 32-⑥, we also check if we need to end the current line and then we output the value of the current node with all extraneous space removed using the `normalize-space` function. This removes all whitespace at the start and at the end of the value and leaves only one space between *non-space* characters.

To output a given number of spaces, we have defined an XPath function called `n1-and-indent` (line 16-④) with one parameter that it used to create a sequence of spaces, which are joined into one string preceded by a newline.

Example 5.9. [compact.xsl]: Stylesheet used to transform the cellar book instance document (Example 2.2) into Example 5.8

```

1 <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:gl="http://www.iro.umontreal.ca/lapalme"
  version="2.0">

5   <xsl:strip-space elements="*" /> ❶
   <xsl:output omit-xml-declaration="yes" method="text" /> ❷

   <xsl:template match="/"> ❸
     <xsl:apply-templates>
10    <xsl:with-param name="indent" select="0" />
     </xsl:apply-templates>
     <xsl:text>
</xsl:text>
   </xsl:template>

15  <xsl:function name="gl:nl-and-indent"> ❹
     <xsl:param name="nb" />
     <xsl:value-of select="concat('&#xA;',
       string-join(for $i in 1 to $nb return ' ',''))" />
20  </xsl:function>

   <xsl:template match="@*"> ❺
     <xsl:param name="indent" />
     <xsl:if test="position()>1">
25     <xsl:value-of select="gl:nl-and-indent($indent)" />
     </xsl:if>
     <xsl:text>@</xsl:text>
     <xsl:value-of select="local-name()" />
     <xsl:text>[</xsl:text><xsl:value-of select="." /><xsl:text>]</xsl:text>
30  </xsl:template>

   <xsl:template match="text()"> ❻
     <xsl:param name="indent" />
     <xsl:if test="count(../*)>0 or position()>1">
35     <xsl:value-of select="gl:nl-and-indent($indent)" />
     </xsl:if>
     <xsl:value-of select="normalize-space()" /> ❼
   </xsl:template>

40  <xsl:template match="*"> ❽
     <xsl:param name="indent" />
     <xsl:if test="count(../*)>0 or position()>1">
       <xsl:value-of select="gl:nl-and-indent($indent)" />
     </xsl:if>
     <xsl:value-of select="local-name()" />
     <xsl:text>[</xsl:text>
45     <xsl:apply-templates select="@*|*|text()"> ❾
       <xsl:with-param name="indent"
         select="$indent + string-length(local-name()+1)" />
50     </xsl:apply-templates>

```

```
        <xsl:text>]</xsl:text>
    </xsl:template>

</xsl:stylesheet>
55
```

- ❶ The XSLT processor will ignore all whitespace nodes from the input so that the output only contains spaces explicitly inserted by the stylesheet.
- ❷ Indicates that the output will be plain text, thus we do not want an XML declaration to be emitted.
- ❸ At the root, starts to apply the compaction algorithm with an indentation of 0. Forces a new line at the end of the output with the content of an `xsl:text` tag.
- ❹ XPath function for outputting a number of spaces given by the `indent` parameter preceded by a new line indicated by a character entity.
- ❺ For an attribute, outputs the name of the attribute preceded by an `@` followed by its value enclosed in square brackets. If this is not the first attribute, indent the following line.
- ❻ For a text node, outputs its normalized value. If there were attributes or if this text node is not the first child, indent the following line.
- ❼ Normalizes a text value, i.e. remove the surrounding spaces and line breaks.
- ❽ For an element, outputs the element name followed by the output of the recursive call to the compaction of attributes, children and text nodes are enclosed in square brackets. If there are any attributes and it this element is not the first one, indent the following line.
- ❾ Updates the indentation for the children nodes with the length of the element name plus one to take into account the open bracket.

5.4. Transformation into PDF with XSL-FO

The previous sections have illustrated the principles of XSLT templates for producing HTML and character output. XSL also defines a more involved and powerful formatting tool: XSL-FO, standing for *eXtensible Stylesheet Language-Formatting Object*. It is similar in principle to the Cascading Style Sheets (CSS) (see next section) defined for HTML to separate the information computation process from the rendering on a specific device (screen, paper, PDA, speech). As shown in the middle of the flow diagram of Figure 1.3 (page 4), Transformations can produce Formatting Objects, i.e. XML elements in the `http://www.w3.org/1999/XSL/Format` namespace with prefix `fo`, which are then rendered on different devices, particularly in PDF.

XSL-FO is a declarative language designed to describe the page content in terms of nested areas, laid out under certain constraints. The main purpose of this approach is the production of printed pages: it allows the definition of the general shape of pages (margins, headers, page numbers, etc.) and the relative placement and nesting of the areas containing the information of the document. Great care has been given to provide a uniform processing of multiple languages and writing systems (not necessarily going from left to right and top to bottom) in the same page. It is also possible to create HTML-like tables and *generalized lists* as pairs of items with aligned labels and bodies. We will use these lists to illustrate the nesting of XML elements in our PDF output as shown at the bottom of Figure 1.5 and in Figure 5.4.

Figure 5.4 shows the three pages generated by the stylesheet of Example 5.10 on Example 2.2. An XML element is displayed with its name in green aligned with its contents. In some cases, the characters of a label overlap but we could not find a reliable way of adjusting the position of a list item body depending on the length of its list item label. We simplify by leaving a distance of 30 mm between the start of the label, given by the element name, and the start of the indented block describing the element value. This limitation is understandable because the relative positions of the label and body must be determined when the `fo` elements are generated by the transformation process but the length of a label is determined when it is rendered on the PDF page.

Figure 5.4. Three pages of PDF output of compaction by Formatting Objects

cellar-book		CellarBook.xsd		Page 1			
wine-catalog	wine	@name	Domaine de l'Île Margaux	country France region Loire Valley producer SCEA Château de Montguéret			
		@appellation	Bordeaux supérieur				
		@classification	a.c				
		@code	C00043125				
		@format	750ml				
		properties	color red alcohol-strength 12.5 nature still				
		comment	Ready for drinking now				
		food-pairing	emph Accompanies Bordelaise ribsteak				
		price	22.80				
		year	2002				
		wine	wine		@name	Fiesling Hugel	country France region Alsace and East producer Hugel & Fils
					@appellation	Alsace	
@classification	a.c.						
@code	C00042101						
@format	750ml						
properties	color white alcohol-strength 12 nature still						
comment	Ready for drinking now. Serve it fresh but not too cold.						
tasting-note	This champagne has a light fruity aroma. It is delicate and has exquisite bubbles.						
price	17.95						
year	2002						
wine	wine			@name	Château Montguéret	country Spain region Old Castille producer Real Sitio de Ventosilla SA	
				@appellation	Anjou		
		@classification	a.c.				
		@code	C10263859				
		@format	750ml				
		properties	color rosé alcohol-strength 11 nature still				
		comment	Ready for drinking now. Serve at 8°-10°C.				
		tasting-note	Tender pink in color, this wine shows light raspberry highlights.				
		price	14.65				
		year	2003				
		wine	wine	@name	Mumm Cordon Rouge		country France region Champagne producer G.H. Martel & Co
				@appellation	Champagne		
@classification	a.c.						
@code	C00312363						
@format	375ml						
properties	color white alcohol-strength 12 nature Champagne						
comment	Ready for drinking now. Serve it fresh but not too cold.						
tasting-note	This champagne has a light fruity aroma. It is delicate and has exquisite bubbles.						
price	33.00						
year	2000						
wine	wine			@name	Prado Rey Roble	country Spain region Old Castille producer Real Sitio de Ventosilla SA	
				@appellation	Ribera-del-duero		
		@classification	d.o.				
		@code	C00929026				
		@format	magnum				
		properties	color red alcohol-strength 12.5 nature still				
		comment	Ready for drinking now. Serve it fresh but not too cold.				
		tasting-note	This champagne has a light fruity aroma. It is delicate and has exquisite bubbles.				
		price	35.25				
		year	2002				

cellar-book		Page 3					
owner	name	first	Jude				
		family	Raisin				
		street	1234 rue des Chateaux				
		city	St-George				
		province	ON				
		postal-code	M7W 7S0				
		location	street	street	4587 des Futailles		
				city	Vallée des crus		
				province	QC		
				postal-code	H3C 4J8		
				cellar	wine	@code	C00043125
						purchaseDate	2005-06-20
quantity	2						
comment	bold Guy Lapalme, Montréal : should reorder soon						
wine	wine					@code	C00312363
						purchaseDate	2004-11-19
						quantity	5
						rating	@stars 3
		comment	Bottle too small...				
		wine	wine			@code	C10263859
						purchaseDate	2005-06-19
						quantity	6
				comment	Really great		
				wine	wine	@code	C00929026
						purchaseDate	2003-10-15
						quantity	1
comment	bold for big parties						

[compactFO-all.pdf]: PDF output of compaction by Formatting Objects (XSL-FO) of Example 2.2. The output spans over three US Letter format pages that we reduced here to show the overview of all pages. The page headers show the context of each page and the page number.

5.4.1. XSL-FO Input to the Renderer

Figure 5.5. Outline of the XSL-FO file produced by running Example 5.10 on Example 2.2

```

2  <fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
3  <fo:layout-master-set>
4  <fo:simple-page-master margin-bottom="1cm" margin-left="1cm" margin-right="1cm"
5  margin-top="1cm" master-name="a-page">
6  <fo:region-body margin-top="1cm" margin-bottom="1cm"/>
7  <fo:region-before extent="1cm"/>
8  </fo:simple-page-master>
9  <fo:page-sequence-master master-name="page-layout">
10 <fo:repeatable-page-master-reference master-reference="a-page"/>
11 </fo:page-sequence-master>
12 </fo:layout-master-set>
13 <fo:page-sequence master-reference="page-layout">
14 <fo:static-content flow-name="xsl-region-before"> [4 lines]
19 <fo:flow flow-name="xsl-region-body">
20 <fo:list-block provisional-distance-between-starts="30mm">
21 <fo:list-item>
22 <fo:list-item-label end-indent="label-end()">
23 <fo:block font-weight="bold" color="green">cellar-book</fo:block>
24 </fo:list-item-label>
25 <fo:list-item-body start-indent="body-start()">
26 <fo:block border-color="black" border-left-style="solid"
27 border-left-width="thin" border-top-style="solid"
28 border-top-width="thin" padding-left="2mm" space-after="1mm">
29 <fo:marker marker-class-name="context"/>
30 <fo:list-block>
31 <fo:list-item> [13 lines]
45 <fo:list-item>
46 <fo:list-item-label end-indent="label-end()"> [3 lines]
50 <fo:list-item-body start-indent="body-start()"> [1863 lines]
1914 </fo:list-item>
1915 <fo:list-item> [116 lines]
2032 <fo:list-item> [65 lines]
2098 <fo:list-item> [517 lines]
2616 </fo:list-block>
2617 </fo:block>
2618 </fo:list-item-body>
2619 </fo:list-item>
2620 </fo:list-block>
2621 </fo:flow>
2622 </fo:page-sequence>
2623 </fo:root>

```

[compactFO.jpg]: Outline of the XSL-FO file produced by running the XSL stylesheet of Example 5.10 on the cellar book example of Example 2.2. This picture was *reduced* with the fold/unfold feature in <oXygen/>.

Because the output of an XSL-FO instance document is processed by another program to get the final PDF output, it is a bit difficult to grasp the processing involved in the XSL-FO workflow. So we will go backwards by first looking at the XSL-FO given as input to the renderer. In principle we could write this XML file by

hand but, looking from Figure 5.5 which shows only the outline of more than 2500 lines for three PDF pages, we appreciate the fact that it can be produced by a machine... The figure is the output produced by the application of the XSL templates of Example 5.10 on the cellar book instance document (Example 2.2).

An XSL-FO file is an XML file starting with a `fo:root` element with two children elements:

- `fo:layout-master-set` that describes the *shape* of the different types of pages that occur in the document and the sequence in which they appear. In Figure 5.5, we have a simple document so
 - `fo:simple-page-master` (lines 4-8) defines a single model for all pages with 1 cm margins. Within it, the header, defined with `fo:region-before` will take the top 1 cm and the *real* content of the page will start another 1 cm lower. Here we define only a single type of master page but for more complex documents it would be possible to have different master page for title pages, for first pages, for even or odd pages, etc.
 - `fo:page-sequence-master` (lines 8-11) declares that the document is an infinite repetition of the above page.
- `fo:page-sequence` (lines 13-2622) defines the content of the document that will be rendered according to the page layout we have defined above. The content of the page is given in the `fo:flow` element (lines 19-2621) which starts in the current page and continues in the *region-body* of the next pages. In fact, the only *visible* text from Figure 5.5 that appears in Figure 5.4 is the first word (`cellar-book`) produced on line 23. Content that appears at the same place within each page, such as headers and footers, is called `static-content` (line 14).

5.4.2. From the Instance Document to the XSL-FO file

We will now look at how to build a stylesheet to produce the XSL-FO file described in the previous section from an XML instance document. Similarly to what we have done to produce HTML output (Section 5.2.3), all tree structures defined with elements with the `fo` namespace prefix will appear verbatim in the output. XSL-FO elements can also be created by `xsl:element` templates but we will not need this here. Example 5.10 starts by defining a group of `xsl:attribute-sets` for defining the global formatting parameters of the file. This set up makes it easier to change the formatting without going into the details of the code.

We must start with a `fo:root` element (line 56-①) with two children:

- `fo:layout-master-set`, within the named template `define-layout` defined on line 32-②, describes the shape of a page with a `fo:simple-page-master` element (line 34-③) that defines its margins relative to the page; within it, we define the *region-body* in which the content will appear; we also define areas for the header (called `fo:region-before`) and the footer (not used in our example). Then the sequence of page masters is given (line 38-④): here a simple repetition of our single page master.
- `fo:page-sequence` (line 58-⑤) refers to a `page-sequence-master` in which the content of the page will be given in the `fo:flow` element (line 60-⑥) which will start in the current page and continue on the next pages within their *region-body*. The content of a header is defined by the named template `define-header` (line 44-⑦) which creates a line with the content of the marker on the left and the page number on the right. As the last (and only) line of the block is to be justified, `fo:leader` will fill the line with white space in between.

The overall tree is defined once for the root element of the document (line 55-⑧) and the traversal of the instance document starts on line 62-⑨ within the `fo:list-block` element in the top-level `fo:flow` element, which corresponds to line 30 of Figure 5.5.

The nested boxes will be at a distance of 30 mm of each other (line 61-⑩). We use the same type of recursive tree traversal algorithm as the one for HTML presentation (Example 5.7) and text compaction (Example 5.9).

Formatting objects create lists as aligned blocks whose relative size and position must satisfy presentation constraints. As can be seen on line 30 of Figure 5.5, a `fo:list-block` (created on line 60-⑨ of Example 5.10) is composed of `fo:list-items` one on top of each other. A `fo:list-item` (line 70-⑦) is composed of a `fo:list-item-label` aligned horizontally with a `fo:list-item-body` (line 76-⑩) even if they are not the same height. In our example, the top and left borders of blocks are colored to show the nesting of blocks which corresponds to the nesting of elements in the XML file. The starting horizontal position of each `fo:list-item-label` is computed from the value of the enclosing block but its end position must be specified. Here it is computed by a predefined function `fo:label-end` which takes into account the value specified for the distance between blocks (line 61-⑩). The starting position of the list item body must also be specified, most often again with a predefined function `fo:body-start` (line 76-⑩).

The processing of elements starts by creating a new `fo:list-item` (line 72-⑥) with the element name in bold green defined in the attribute set on line 27-④ as `fo:list-item-label` (line 70-⑦). The `fo:list-item-body` (line 76-⑩) processing depends on whether there are any children elements (or attributes) or not:

- when there are no children elements (line 78-⑫) but possibly text nodes, we display the content of text nodes in a `fo:block`.
- when there are children nodes (line 85-⑭), they are displayed within a bordered `fo:block` whose content is a recursively built (line 92-⑰) `fo:list-block`. As we explain below, the current context is also computed and saved in the `fo:marker` (line 87-②).

Attributes (line 101-⑳) are displayed using the `labeledValue` named template (line 128-㉑): their name is in blue italicized text (as the `fo:list-item-label`) and their text content as `fo:list-item-body`. Note here the use of a tree fragment value as actual parameters (line 104-㉓ and line 108-㉕) to the `labeledValue` named template. `fo:inline` elements create *ordinary* text. The area for display is created with a `fo:block` element in the `labeledValue` named template (line 128-㉑). This template creates a `fo:list-item` comprising a `fo:list-item-label` and a `list-item-body`. In both cases (line 134-㉗ and line 139-㉙), we insert the tree given as parameter by means of `xsl:copy-of` and not the *usual* `xsl:value-of`, which would only return the *content* of the tree given as parameter and not the tree value itself.

Text nodes (line 116-㉚) are output as a list item with an empty label i.e. an empty in-line block, again using the `labeledValue` named template, and the text content as body for the list item. Text nodes comprising only spaces and newlines (whose normalization gives an empty string) are ignored.

Because the output of this program is longer than a single page (see Figure 5.4), then it *flows* on the following one. But it is interesting to show the current context of the start of the page in its header. This is done by creating a `fo:marker` (line 87-②) at each new nested block containing the names of the elements that are the ancestors of the current node. The current value of a marker at the start of the page will appear in the left part of the header (line 47-①).

Example 5.10. [compactFO.xsl] Stylesheet to transform the information of the cellar book (Example 2.2) into the colored nested blocks representation of Figure 5.4

```

1  <?xml version="1.0" encoding="UTF-8"?>
   <xsl:stylesheet xmlns:fo="http://www.w3.org/1999/XSL/Format" version="2.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

5   <xsl:attribute-set name="page-size">                                ❶
     <xsl:attribute name="margin-top">1cm</xsl:attribute>
     <xsl:attribute name="margin-left">1cm</xsl:attribute>
     <xsl:attribute name="margin-right">1cm</xsl:attribute>
     <xsl:attribute name="margin-bottom">1cm</xsl:attribute>
10  </xsl:attribute-set>

   <xsl:attribute-set name="block-decoration">                        ❷
     <xsl:attribute name="border-color">black</xsl:attribute>
     <xsl:attribute name="border-left-style">solid</xsl:attribute>
15  <xsl:attribute name="border-left-width">thin</xsl:attribute>
     <xsl:attribute name="border-top-style">solid</xsl:attribute>
     <xsl:attribute name="border-top-width">thin</xsl:attribute>
     <xsl:attribute name="padding-left">2mm</xsl:attribute>
     <xsl:attribute name="space-after">1mm</xsl:attribute>
20  </xsl:attribute-set>

   <xsl:attribute-set name="element-formatting">                    ❸
     <xsl:attribute name="font-weight">bold</xsl:attribute>
     <xsl:attribute name="color">green</xsl:attribute>
25  </xsl:attribute-set>

   <xsl:attribute-set name="attribute-formatting">                  ❹
     <xsl:attribute name="font-style">italic</xsl:attribute>
     <xsl:attribute name="color">blue</xsl:attribute>
30  </xsl:attribute-set>

   <xsl:template name="define-layout">                                ❺
     <fo:layout-master-set>
       <fo:simple-page-master master-name="a-page" xsl:use-attribute-sets="pa
35       <fo:region-body xsl:use-attribute-sets="page-size" margin-left="0cm"
         <fo:region-before extent="1cm"/>
       </fo:simple-page-master>
       <fo:page-sequence-master master-name="page-layout">          ❷
         <fo:repeatable-page-master-reference master-reference="a-page"/>
40       </fo:page-sequence-master>
     </fo:layout-master-set>
   </xsl:template>

   <xsl:template name="define-header">                                ❸
45     <fo:static-content flow-name="xsl-region-before">
       <fo:block text-align-last="justify">
         <fo:retrieve-marker retrieve-class-name="context"           ❹
           retrieve-position="first-starting-within-page"/>
         <fo:leader/>

```

```

50         Page <fo:page-number/>
           </fo:block>
         </fo:static-content>
       </xsl:template>

55     <xsl:template match="/">                                10
       <fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format"> 11
         <xsl:call-template name="define-layout"/>
         <fo:page-sequence master-reference="page-layout">      12
           <xsl:call-template name="define-header"/>
           <fo:flow flow-name="xsl-region-body">                13
             <fo:list-block provisional-distance-between-start 14s="30mm">
               <xsl:apply-templates/>                            15
             </fo:list-block>
           </fo:flow>
         </fo:page-sequence>
       </fo:root>
     </xsl:template>                                          16

<xsl:template match="*">
70   <fo:list-item>                                          17
     <fo:list-item-label end-indent="label-end()">
       <fo:block xsl:use-attribute-sets="element-formatting"> 18
         <xsl:value-of select="local-name()"/>
       </fo:block>
     </fo:list-item-label>
75   <fo:list-item-body start-indent="body-start()">        19
     <xsl:choose>
       <xsl:when test="count(*)=0 and count(@*)=0">          20
         <fo:block>
80           <fo:inline font-style="normal" color="black">
               <xsl:value-of select="."/>
             </fo:inline>
           </fo:block>
         </xsl:when>
85         <xsl:otherwise>                                    21
           <fo:block xsl:use-attribute-sets="block-decoration">
             <fo:marker marker-class-name="context">          22
               <xsl:value-of select="string-join(for $n in ancestor::*
                 return local-name($n),' | ')">
             </fo:marker>
           <fo:list-block>
             <xsl:apply-templates select="@*|*|text()"/>      23
           </fo:list-block>
         </fo:block>
       </xsl:otherwise>
     </xsl:choose>
   </fo:list-item-body>
 </fo:list-item>
</xsl:template>

100  <xsl:template match="@*">                                24
     <xsl:call-template name="labeledValue">
       <xsl:with-param name="label">

```

```

105     <fo:inline xsl:use-attribute-sets="attribute-formatting"> ❸
        @<xsl:value-of select="local-name()" />
    </fo:inline>
    </xsl:with-param>
    <xsl:with-param name="value"> ❹
        <fo:inline font-style="normal" color="black">
110     <xsl:value-of select="." />
        </fo:inline>
    </xsl:with-param>
    </xsl:call-template>
</xsl:template>

115 <xsl:template match="text()"> ❺
    <xsl:variable name="content" select="normalize-space(.)" />
    <xsl:if test="string-length($content)>0">
    <xsl:call-template name="labeledValue">
120     <xsl:with-param name="label">
        <fo:inline/>
    </xsl:with-param>
    <xsl:with-param name="value" select="$content" />
    </xsl:call-template>
125 </xsl:if>
</xsl:template>

    <xsl:template name="labeledValue"> ❻
    <xsl:param name="label" />
130 <xsl:param name="value" />
    <fo:list-item>
    <fo:list-item-label end-indent="label-end()">
    <fo:block>
135     <xsl:copy-of select="$label" />
    </fo:block>
    </fo:list-item-label>
    <fo:list-item-body start-indent="body-start()">
    <fo:block>
140     <xsl:copy-of select="$value" />
    </fo:block>
    </fo:list-item-body>
    </fo:list-item>
    </xsl:template>

145 </xsl:stylesheet>

```

- ❶ Attributes that define the global margins of the document.
- ❷ Attributes that define how the box surrounding an element will be displayed.
- ❸ Attributes for the formatting of an element name.
- ❹ Attributes for the formatting of an attribute name.
- ❺ Creates the global layout, here we have the same layout for each page.
- ❻ Defines the layout of a page with margins of 1cm so that they coincide with the global margins.
- ❼ Defines the global layout of pages as the repetition of the same page master.

- ⑧ Define the content of the header, it appears in the `region-before` as the value of the `context` marker and the page number. The combination of the `text-align-last` attribute and `fo:leader` ensures that enough space is added so that the page number appears right-aligned at the right margin of the page.
- ⑨ Gets the current value of the marker defined on line 87-②.
- ⑩ Template that matches the root node and that defines the overall shape of the XSL-FO output enclosed in a `fo:root`.
- ⑪ Defines the layout by calling the template defined at line 32-⑤ followed by the sequence of pages.
- ⑫ The page sequence is filled out according to the definition of pages given on line 38-⑦. It first defines the content of the header by calling the named template on line 44-⑧ and then gives the content of the document in a single flow.
- ⑬ The content of the document to appear in the `region-body` is composed of a single list-block.
- ⑭ Specifies the distance between the start of its item and the start of its body.
- ⑮ The content of a `fo:list-block` is filled by the recursive traversal of all elements of the source document and the application of the appropriate templates depending on the types of elements.
- ⑯ Template for an element.
- ⑰ Creates an `fo:list-item` having the name of the element as label. The body of the `fo:list-item` is the content of the element.
- ⑱ Outputs the element name with the formatting defined on line 22-③.
- ⑲ Outputs the content of an element depending on whether it has any children or attributes.
- ⑳ When there are no children nodes, the content of text nodes are copied.
- ㉑ When there are children nodes or attributes, creates an internal block. First updates the value of the `context` marker with a string built from the ancestors of this node. This string will be used in the page header (line 44-⑧). Then starts a new `fo:list-block` to be filled by a recursive application of the templates on the content of all nodes.
- ㉒ Creates the content of the marker with an XPath expression that loops over all ancestors of the current node and creates a list with the names of their elements. The strings in this list are then concatenated separated with a vertical bar.
- ㉓ Applies the templates on all attributes, element and text nodes of this element.
- ㉔ An attribute is output as a one-element list using the named template defined on line 128-⑳. Note that the parameter values are complex XSL-FO elements.
- ㉕ The `label` parameter is the name of the attribute preceded by an `@` with the appropriate formatting defined on line 27-④.
- ㉖ The `value` parameter is the value of the attribute (a text node).
- ㉗ A non-empty text node is output as a one-element list using the named template defined on line 128-㉓ with an empty label and the normalized text content as `value` parameter.
- ㉘ Named template for outputting a single list item with the content of two formal parameters.
- ㉙ Because the actual parameter corresponding to `label` can be a complex XSL-FO element, we need to copy the whole tree of the parameter and not simply use `xsl:value-of`.
- ㉚ Because the actual parameter corresponding to `value` can be a complex XSL-FO element, we need to copy the whole tree of the parameter and not simply use `xsl:value-of`.

This section has shown how to produce *publication quality* output from an XML file. We have used nested list blocks to align the name of an element with its content, but nested tables could also have been used. This would allow the horizontal centering of the element name with respect to its content. The principles remain the same but the code would be a bit longer because tables have more options.

5.5. Transformation with a Cascading Style Sheet (CSS)

Another type of stylesheet are Cascading Style Sheets (CSS) [11] that allow authors and users to attach font and spacing information to structured documents such HTML and XML documents. By separating the presentation style of the documents from their content, CSS simplifies Web authoring and site maintenance. Although it is not an XML technology per se, it bears some resemblance with XSL. Sharing the same name often brings some confusion between them so we think it is useful to give an example of what type of *compact form* can be achieved with a CSS. We will then compare this tool with what has been done with XSLT and XSL-FO.

As we said in the previous section, CSS and XSL-FO share some concepts and terminology about formatting. Both use a *nested box* modeling approach and many formatting properties sharing the same name. A CSS is set of formatting rules that can be defined at the level of the tag itself, the document or even of a web site. A rule defines if the information within the tag should be displayed as a *block* starting a new line or if it should be inserted *in-line*; we have seen a similar distinction in XSL-FO. The rule can further define the formatting attributes (fonts and color), the padding, the margins and the borders around the information within this tag; again the same terminology can be found in XSL-FO.

The formatting to be applied on an element depends on whether an element corresponds to the selector of a rule. Selectors can depend on the name of the element, its class (the value of the `class` or `id` attribute), its position in the document (e.g. whether is a child or sibling of other elements). It is also possible to check the value of its attributes. It is also possible to have a different formatting if the user clicks or hover the mouse on an element or if it is a link that has been visited; see [57] for more details. There is a very limited way of generating some content either before or after the output of the content of the tag, see Example 5.11.

Compared with XSLT, the formatting that can be obtained with CSS is relatively limited because it is not possible to compute new information or to change the order of the elements in the file. So the information given in the tags and attributes can only be used for labelling the text nodes and cannot have any informational content that can appear in the output. This is the case in HTML files in which tags are only used for formatting. The only effective control that can be achieved with CSS is the selective display the information either by leaving some space or by removing it completely from the display. This is quite rudimentary compared with what can be done with XSLT. But when CSS are combined with Javascript, it is possible to build interactive displays such as tooltips or have some parts that appear or disappear when the user clicks on certain parts of the display. It is also possible to add some content before or after the text content of the element, but this is a bit awkward and no computation of this new information is possible. But it must be remembered that CSS are designed as a declarative formalism for separating formatting information from content and not for creating new information. One way of dynamically creating new content in a web page is the use of the AJAX technologies, discussed in Chapter 10, that allow the modification of the underlying document structure.

Example 5.11. [`compact.css`] Cascading Style Sheet for displaying the content of the cellar book in a web page

The last 25 lines of the file are not shown as they are similar to the two last ones displayed here (the name of the tag is given as content).

```

1 * {
    display:block;
    position:relative;
    margin-left: 20px;
5   border-color:black;
    border-left-style: solid;
    border-left-width: thin;
    border-top-style: solid;
    border-top-width: thin;
10  padding-left: 2mm;
    padding-bottom: 1mm;
    }

    *:before {
15   font-weight: bold;
    color:green;
    }

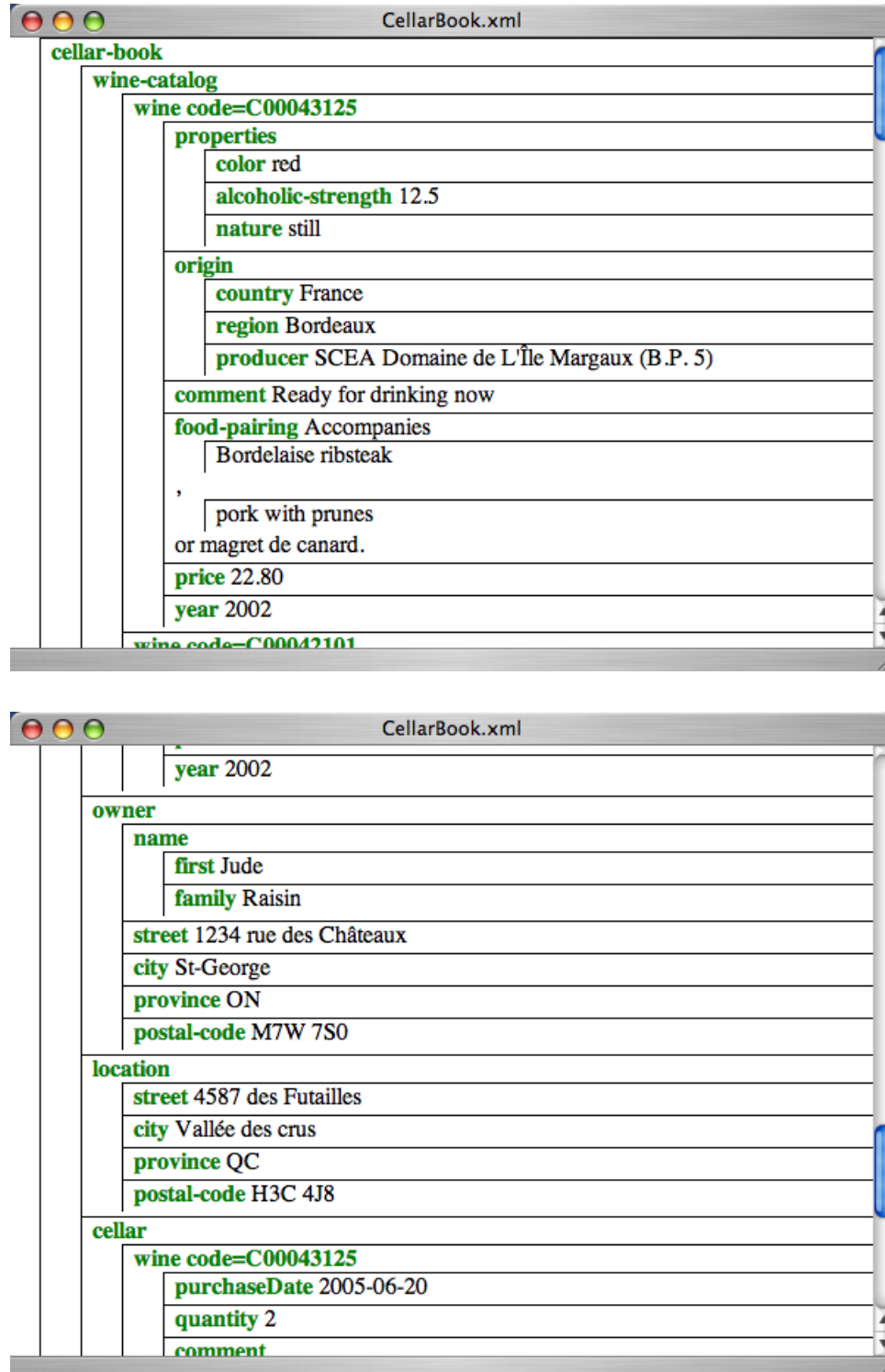
    cellar-book:before {content:"cellar-book "};
20 wine-catalog:before {content:"wine-catalog "};
    wine:before {content:"wine code="attr(code);}
    properties:before {content:"properties "};
    color:before {content:"color "};
    ...
25

```

- ❶ Rule for any element: it is displayed as a block relatively placed at 20 pixels to the left of its enclosing block; the borders and padding are the same as for `block-decoration` attribute set in line 12-❷ of Example 5.10.
- ❷ All generated text that will appear before a tag will be in bold green.
- ❸ As there is no way of accessing the name of the current tag, a specific rule must be defined for each type of tag to insert its name in front of its content.
- ❹ The `attr` function returns the value of a specific attribute. So, after the name of the tag `wine` followed by the name of the attribute, we output the value of the attribute `code`. As a CSS does not deal with namespaces, this rule is used for both the `wine` elements in the catalog and the ones in the cellar.
- ❺ 25 lines that follow the same pattern as the two previous ones: `tag-name :before {content:"tag-name "};`

Figure 5.6. CSS formatting of Example 2.2 with the stylesheet of Example 5.11

Two excerpts of the display in a web browser of the cellar book using the CSS stylesheet. The nested box display is to be compared with the one shown in Figure 5.4. The parts in bold (green if seen in color) are *generated content*.



5.6. Associating an Instance File to a Stylesheet

Transforming an instance file with a stylesheet is most often performed by externally specifying the transformation stylesheet file to apply to a given instance file. This can be achieved using an XML editor in which we can associate an XML file with a stylesheet (and vice versa). Some editors also allow the definition of many transformation scenarios. The transformation can also be done in *batch* mode by specifying a stylesheet to a transformation engine. For example, in the companion website of this report, we show a simple Java program which can be used as a Unix filter to standard input. We can get on the standard output the compact text output of the wine catalog with the following call:

```
java Transform compact.xsl < WineCatalog.xml
```

This program also allows to specify run-time parameters to the stylesheet (declared with top-level `xsl:param` elements as shown in Section 5.2.1). To get an HTML file with the table of the *white* wines of the catalog (similar to Example 5.1), we can use the following:

```
java Transform WineCatalog.xsl color white < WineCatalog.xml > whites.html
```

It is also possible to use other transformation program directly from the command line `xsltproc` (but it currently deals only with XSL 1.0 stylesheets), `SAXON` or `XALAN` for which it is necessary to set your `CLASSPATH` appropriately.

```
xsltproc --stringparam color white WineCatalog.xsl WineCatalog.xml > whites.html
```

```
java net.sf.saxon.Transform -s WineCatalog.xml WineCatalog.xsl color=white \  
> whites.html
```

```
java org.apache.xalan.xslt.Process -XSL WineCatalog.xsl -IN WineCatalog.xml \  
-OUT whites.html -PARAM color white
```

Because stylesheets can also be interpreted by web browsers, it is also possible to link an instance file directly to a stylesheet by means of the `xml-stylesheet` processing instruction. For example, if one adds the following at the beginning of Example 2.3

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

then, upon loading the XML file `WineCatalog.xml` into a web browser, the catalog will be displayed after the transformation defined by Example 5.7. But one must be careful before relying on such *automatic* transformations, because not all browsers implement all XSL transformations, especially those of XSLT 2.0.

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

is the way to make link with a CSS in an XML or HTML page. Even though all browsers implement CSS formatting, there are still some inconsistencies between platforms.

5.7. Additional Information on XSL

The official information on Extensible Stylesheet Language (XSL) [7] is comprehensive and detailed on making it more than 400 printed pages, the first 50 pages describe the basic principles of the tree transformations that lead to formatting. The remaining pages describe all the possible options for all parameters. One should also consult the XSLT language description [28] and because the XSL description takes it for granted.

- **Transformation**

- <http://www.w3.org/Style/XSL/> is the best starting point to get information on XSLT with links to tools and tutorials.
- <http://www.mulberrytech.com/quickref/XSLTquickref.pdf> is a nice XSLT and XPath Quick Reference (US legal size)
- <http://www.iro.umontreal.ca/~lapalme/ForestInsteadOfTheTrees/QuickRefSheets/xslt2quickref.pdf> XSLT 2.0 Quick reference sheet (2 pages - US letter size)
- <http://www.iro.umontreal.ca/~lapalme/ForestInsteadOfTheTrees/QuickRefSheets/xslt2quickrefAbridged.pdf> Abridged XSLT 2.0 Quick reference sheet showing only the most often used elements and attributes (1 page US letter)
- <http://www.dpawson.co.uk/xsl/> is another very useful site with a lot of practical informations about XSL
- <http://xml.apache.org/xalan-j/> Xalan is a public domain stylesheet processor that works in conjunction with Xerces [4]

- **Formatting objects**

- <http://www.ibiblio.org/xml/books/bible2/chapters/ch18.html> is a very good initial tutorial on Formatting Objects
- XSL Formatting Objects Developer's Handbook[31] is a very thorough and didactic introduction and reference on Formatting Objects
- <http://xmlgraphics.apache.org/fop/> FOP is a public domain Formatting object renderer written in Java; it still has a few limitations with respect to the official standards.
- <http://www.renderx.com> RenderX is selling XEP, written in Java, a commercially available XSL-FO rendering engine implementing the official specification. It was used to produce the PDF version of this document. An academic license agreement is available. They also publish a tutorial, which is a very good starting point for learning XSL.

- **Cascading Style Sheets**

- *Beginning CSS: Cascading Style Sheets for Web Design* [57] A thorough description of CSS with many practical examples. There are also tables showing the implementation status of CSS features in different browsers.

Chapter 6. Document Query

In the previous chapter, we described how to transform an XML file into another by means of tree transformations defined by templates. We have also seen how to extract information from an XML file. But now that XML files are used for keeping data similarly to databases, an alternate way of querying XML information has been defined, resulting in a language close to the well-known SQL.

It is called XQuery and is aimed at selecting information from XML databases that are often too large to be stored in a single XML file for which a random access would not necessarily be efficient. XQuery can also easily combine information from many XML documents, which is a bit awkward to do in XSLT. The output of an XQuery query is a sequence of XML nodes that lend themselves to the full power of XPath functions and XML constructors in order to further convert them into other XML nodes.

For the simple use cases we have shown in the previous chapter, transforming an XML document into an XHTML one by means of XSLT templates can also be seen as querying an input document to select some information which is then inserted into an XHTML template. This chapter illustrates some simple uses of XQuery to solve the problems shown previously.

Contrarily to XML Schema and XSLT, XQuery is not a fully XML based notation. Some readers might find it less verbose or rebarbative than XSLT. Relax NG Compact notation is also a non-XML-based notation for schemas of XML files when compared to XML Schema.

An XQuery *program* is called a **query**. It is an expression that selects or constructs a sequence of XML nodes. A query can take one of the following forms:

- The simplest kind of query is an XPath 2.0 expression returning a sequence of nodes, but this is limited to the extraction of XML information from the input document without change. XQuery can use all XPath 2.0 functions (in fact, both XQuery and XPath functions are defined in the same document [33]). When applied to our cellar-book instance document (Example 2.2), the following query (the same as line 2-2 of Example 4.1) returns the sequence of wine elements for which there are less than 2 bottles left in the cellar. Note that the less-than sign can be used without having to type the `<` entity because an XQuery query is not an XML expression.

```
/cellar-book/cellar/wine[quantity<2]
```

- A more powerful type of query is a **FLOWR** expression¹ (for, let, where, order by, return) which is patterned after the SQL `SELECT - FROM - WHERE` instruction. It allows the reordering of results, the creation of new values, etc.

```
for $w in /cellar-book/cellar/wine,  
    $cat-w in /cellar-book/cat:wine-catalog/cat:wine  
    let $q := $w/quantity  
    where $w/@code = $cat-w/@code  
    order by $q  
return concat($q, ":", $cat-w/@name)
```

Like in XSLT, variables must be prefixed by a dollar sign to differentiate them from element names. A FLOWR query bears some resemblance with the `for` loop in XPath 2.0. The `for` in this query (equivalent

¹pronounced *flower* even though the letters are not in the proper order..

to line 11-❶ of Example 4.1) with its two embedded loops performs a join between all the wines in the cellar and the ones of the catalog; `let` creates a variable `$q` to keep track of the wine quantity in the cellar; `where` keeps only wines sharing the same `code` attributes in the cellar and in the catalog; `order` ensures that the output will be in increasing order of quantity; `return` builds the resulting string composed of the string value of `$q` and the name of the wine separated by a colon.

- An XML element that can be created either directly by writing an XML tag in the query (this is called a *direct* constructor) or by using a *computed* constructor. The following query uses two direct constructors: `cheap-wines` and `cheap-wine`. Braces within a direct constructor are used to embed non-XML XQuery code. The following example extracts French wines from the catalog that cost less than 20 dollars (like line 14-❷ of Example 4.1). It creates as output a single XML element called `cheap-wine` containing a list a wines for which we have changed the `cat:wine` tag by `cheap-wine`. The attributes and elements of `cat:wine` are copied to the `cheap-wine` with another nested XPath expression.

```
<cheap-wines>
{for $w in /cellar-book/cat:wine-catalog/cat:wine
  where $w/cat:origin/cat:country='France' and $w/cat:price < 20.0
  return <cheap-wine>{$w/@*, $w/*}</cheap-wine>}
</cheap-wines>
```

The next query is equivalent to the previous one but uses computed constructors. A computed constructor is created by an `element` followed by an identifier and an expression within braces. Attributes can be similarly constructed with an `attribute` construct. The element or attribute identifier can also be computed by writing an expression within braces instead of an identifier.

```
element cheap-wines{
  for $w in /cellar-book/cat:wine-catalog/cat:wine
  where $w/cat:origin/cat:country='France' and $w/cat:price < 20.0
  return element cheap-wine{$w/@*, $w/*}
}
```

- A sequence of the above XQuery expressions separated by commas, parentheses can be used to nest sequences within others.

A query is often preceded by a "prolog" composed of declarations separated by a semicolon. For example, in the previous examples, the query should have been preceded by

```
declare namespace cat = "http://www.iro.umontreal.ca/lapalme/wine-catalog";
```

in order to define the `cat` namespace prefix used in the query. Declarations can also be used to define variables, functions and other options to control the processing of the query. Most often the prolog is much longer than the query itself as there can be only one query in a file. This is not really a limitation because multiple queries can be merged into a single one: one only needs to create a sequence of queries by separating them with commas.

For complete details on the syntax of a query, see EBNF grammar of XQuery.

6.1. XQuery output in HTML

6.1.1. Table

XQuery is designed to extract parts of an XML file. It is therefore quite appropriate for selecting a subset of wines and displaying it as an HTML page.

Selecting only red wines in our wine catalog (Example 2.3 (page 12)) and outputting an HTML table of a subset of the available information for each (Figure 5.1) can be achieved with the XQuery query given in Example 6.1. In order to be compared to Example 5.2, we deliberately organized the XQuery script with a structure similar to the one used in the XSL stylesheet.

The query in Example 6.1 first defines a function for the root node (line 13-④). Because the wine catalog is defined in a specific namespace, its prefix must be declared (line 1-①) and used for selection. This function outputs the overall structure of the XHTML file with direct constructors. Within an enclosed expression, it calls the `wine-catalog` function to create the content of the body. For comparison purposes, on line 23-⑤ we show an alternate way of defining the `root` function using computed constructors instead of giving the HTML structure. The `wine-catalog` function (line 30-⑥) outputs a global heading and then starts a table and defines its headers. The lines of the table will be filled by selecting (line 37-⑦) wines whose color property is `$color`. To set up this color filter, a value must be assigned to the global `color` external variable (line 4-②). This value is set in an implementation-dependent way when the query is executed by the XQuery processor.

The output for each selected wine is defined with a function that is called for each `cat:wine`. It outputs, on a single row of the table, the values of its attributes, its color, its year (right-aligned) and formats its price to start with a dollar sign (right-aligned). It finally renders the volume of each bottle in milliliters and in liters. Because the information in the wine catalog is not stored in milliliters, we call two local functions to transform it appropriately.

Example 6.1. [WineCatalog.xq] XQuery script to select the red wines in the catalog (Example 2.3) and to produce Example 5.1 displayed as Figure 5.1. Compare this with Example 5.2.

```

1 declare namespace cat = "http://www.iro.umontreal.ca/lapalme/wine-catalog"①;
  declare default element namespace "http://www.w3.org/1999/xhtml" ;

  declare variable $color external;                                ②

5  (: to produce legal and validable XHTML ... :)                ③
  declare option saxon:output "method=xml" ;
  declare option saxon:output "doctype-public=-//W3C//DTD XHTML 1.0 Strict//EN" ;
  declare option saxon:output "doctype-system=http://www.w3.org/TR/xhtml1/DTD/xhtml1-
10 declare option saxon:output "indent=yes" ;

  (: using a direct element constructor, with an enclosed expression :)
  declare function local:root($catalog) {                          ④
    <html>
15      <head><title>Wine Catalog</title></head>
      <body>

```

```

        {local:wine-catalog($catalog)}
    </body>
</html>
20 };

(:alternate definition of the function above using computed constructors :)
declare function local:root-bis($catalog){
    element html {
25         element head {element title {"Wine Catalog"}},
            element body {local:wine-catalog($catalog)}
    }
};

30 declare function local:wine-catalog($catalog){
    element h1 {concat("Wine Catalog (", $color, " only)"),
    element table {
        attribute border {1},
        element tr {
35             for $t in ('Wine Name', 'Code', 'Color', 'Year', 'Price', 'ml', 'l')
                return element th{$t}},
            for $wine in $catalog/cat:wine
                where $wine/cat:properties/cat:color=$color
                return local:wine($wine)
40         }
    };

    declare function local:wine($wine){
        <tr>{
45         <td>{data($wine/@name)}</td>,
            <td>{data($wine/@code)}</td>,
            <td>{data($wine/cat:properties/cat:color)}</td>,
            <td align="right">{data($wine/cat:year)}</td>,
            (: format-number is not available in XQuery 1.0 !! :)
50         <td align="right">{concat('$', $wine/cat:price)}</td>,
            <td align="right">{local:toML($wine/@format)}</td>,
            <td align="right">{local:toL($wine/@format)}</td>
        }
    }
};

55 };

    declare function local:toML($fmt){
        if ($fmt='375ml') then '375'
        else if ($fmt='750ml') then '750'
60         else if ($fmt='1l') then '1000'
            else if ($fmt='magnum') then '1500'
            else 'big'
    };

65 declare function local:toL($fmt){
    let $ml := local:toML($fmt)
    return
        if ($ml castable as xs:integer)
            then number($ml) div 1000
70         else $ml
};

```



```
};  
  
local:root(/cat:wine-catalog)
```

12

- ❶ Definition of the `cat` namespace prefix in order to access the elements of the wine catalog defined in this namespace. The default namespace is set to be the one needed for a valid XHTML file. With this declaration, HTML tags that are used in the stylesheet are in the appropriate namespace for HTML validation.
- ❷ Global variable that should be initialized when executing the query.
- ❸ Declarations telling the SAXON transformation engine to serialize with the appropriate headers to produce valid XHTML.
- ❹ Function called on the root node defining the skeleton of the HTML page: a head and a body with a call to a function filling the content of the HTML skeleton.
- ❺ Alternate definition of the previous function to illustrate the difference in style between the direct and computed constructors.
- ❻ Function for the catalog node: a header with a title and a table.
- ❼ Uses a `for-each` in a list of strings to output the headers of the table.
- ❽ A `for-where` instruction selects the wines with the chosen `color`. The result is a sequence `wine` nodes. The function on line line 43-❾ is applied to each of them.
- ❾ Outputs properties of a wine. The first four are written as they appear in the source but we must use the `data` function to keep only the string value. The `price` is formatted in dollars and cents and the bottle format is given in either milliliters or liters through local functions.
- ❿ a multi-line XPath expression to select the appropriate case within a cascaded `if-then-else` expression.
- ⓫ Uses the output of the previous function to get the number of milliliters, which is then divided by 1000 if it is a number. Otherwise it is returned as is.
- ⓬ Query the document starting from the root element node.

6.1.2. Computing New Information

Here we show how XQuery can be used for more complex selections and transformations. We will explain how to create a web page presenting the content of the cellar and integrating information from the wine catalog. This is equivalent to the program shown in Example 5.4. The end result is shown in Figure 5.2 (an outline of the underlying HTML code is shown in Example 5.3). There are external links in order to retrieve more information about the wines by *googling* the name of the wine. There are two similar links for each wine, created for the sole purpose of comparing ways of creating them in XSL.

The `root` function (line 11-❷) creates the high-level structure of the XHTML file. The title of the page, also displayed at the top of the page, refers to the name of the owner. In Example 2.2, element name (line 12-❸) is subdivided in two elements: `first` and `family`. When the value of such an element is accessed in an enclosed expression, it is returned as an XML element. This would not be valid in an XHTML document, so we call the `data` function to get the text content of this element, including the whitespace nodes.

The content of the cellar book is obtained by the `cellar-book` function on line 20-❹ creating a table with the address of the owner (line 28-❺) and the cellar (line 31-❻). It then calls the `cellar` function (line 37-❼). The lines of the addresses are obtained by looping over all elements with a `for` and writing a `
` between the text values of each element of the `owner` and `location` elements. Because we want to skip

the first element (the name of the owner has already been given at the top), we only keep elements (line 28-④) with a position number greater than 1.

The `cellar` (line 40-⑦) function produces a table of information about wines in the cellar, sorted by their code. This is why the `for` is followed by an `order by` specification. The last line of the table contains an estimated total value of the cellar. The value of each type of wine is computed by creating a sequences of values obtained by a `join` (two nested expressions on a `for`) between the codes of wines in the cellar and the codes of the wines in the catalog. The values are then summed with the predefined `sum` function. To compute the total number of bottles (line 60-⑩), we can use the `sum` function. Finally, if there are any comments in the wine elements of the cellar (line 65-⑪), we add a `Comments` section and write each of them, also in increasing order of wine code. This way, they are in the same order as in the table of wines.

We define the `wine` function (line 75-⑫) to create a row in the table of wines. We first define a variable `$code` (line 76-⑬) holding the value of the `code` attribute. In order to retrieve some information about this wine from the wine catalog, we pass the wine node from the wine catalog as a parameter when calling the `nameAndUrl` function (line 80-⑭). The link between the current element and the corresponding element in the catalog is made using the value of the `code` variable given in the XPath expression. The remaining elements of the row are the purchase date (right-aligned), a number of stars corresponding to the rating and the quantity (right-aligned). The estimated value of the cellar is computed using XPath expressions. Note the use of the `data` function to get the value of the elements.

`nameAndUrl` is a function that receives a wine element as a parameter. From this wine element, it creates an XHTML link with an `a` element. Its `href` attribute value is a string specifying the query to send to Google when searching for this wine using its name. Because the link must be created dynamically, we use a computed attribute constructor within a computed `a` element which uses the `data` function to get the value of the `name` attribute. Note that if only `$wine/@name` had been specified, then the **attribute** name would have been added to the `a` element and not its value as content of the element.

The `comment` function (line 97-⑯) outputs the value of the `code` of the parent node, then a colon and the text value of the comment, followed by a line break. When getting the content of the comment, the `cat:bold` elements will be processed by the `bold` function (line 101-⑰). It will transform them in HTML `b` tags. In XSL, this call was implicit via the application of templates but in XQuery the function must be called explicitly.

Example 6.2. [CellarBook.xq] XQuery script to produce information about the cellar (Example 2.2). The resulting HTML code (Example 5.3) is rendered as Figure 5.2. Compare this with Example 5.4.

```

1 declare namespace cat = "http://www.iro.umontreal.ca/lapalme/wine-catalog";

   (: to produce legal and validable XHTML ... :)
   declare option saxon:output "method=xml";
5  declare option saxon:output "doctype-public=-//W3C//DTD XHTML 1.0 Strict//EN";
   declare option saxon:output "doctype-system=http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict";
   declare option saxon:output "indent=yes";

   declare variable $GoogleStart := "http://www.google.com/search?q="; ①
10  declare function local:root($cellar-book){                               ②
       <html>

```

```

        <head><title>Cellar of {data($cellar-book/owner/name)}</title></head>
        <body>
15         {local:cellar-book($cellar-book)}
        </body>
    </html>
};

20 declare function local:cellar-book($cellar-book){
    <h1>Cellar of {data($cellar-book/owner/name)}</h1>,
    <table border="1">
        <tr>
            <th>Personal address</th>
25         <th>Cellar address</th>
        </tr>
        <tr><td>
            {for $e in $cellar-book/owner/*[position()>1]
            return (data($e),<br/>)}
30         </td>
            <td>{for $e in $cellar-book/location/*
            return (data($e),<br/>)}
            </td>
        </tr>
35     </table>,
    <p/>,
    local:cellar($cellar-book/cellar,$cellar-book/cat:wine-catalog)
};

40 declare function local:cellar($cellar,$catalog){
    <table border="1">
        <tr>{
            for $t in ("Code","Name","Purchase Date","Rating","Nb Bottles")
            return <th>{$t}</th>
45     </tr>{
        for $w in $cellar/wine
            order by $w/@code ascending
            return local:wine($w,$catalog)}
        <tr>
50     <td colspan="3">Estimated value</td>
        <td align="right">{
            (: compute the estimated value of the cellar :)
            sum (for $w in $cellar/wine,
                $codeInCat in $catalog/cat:wine/@code
90             where $codeInCat = $w/@code
                return $w/quantity * $codeInCat/../../cat:price)}
60     </td>
        <td align="right">{
            (: compute the total number of bottles :)
            sum($cellar/wine/quantity)}
            </td>
        </tr>
    </table>,
    (: put comment section if at least one comment appears :)
65     if (count($cellar/wine/comment)>0)
        then (

```

```

        <h3>Comments</h3>,
        <p>{for $c in $cellar/wine/comment
            order by $c/../@code ascending
70         return local:comment($c)}
        </p>
    ) else ()
};

75 declare function local:wine($wine,$catalog){
    let $code := $wine/@code
    return
        <tr>
            <td>{substring($code,2)}</td>
80         <td>{local:nameAndUrl($catalog/cat:wine[@code=$code])}</td>
            <td align="right">{data($wine/purchaseDate)}</td>
            <td align="center">{substring('*****',1,
85         if ($wine/rating/@stars) then $wine/rating/@stars else 0)}
            <td align="right">{data($wine/quantity)}</td>
        </tr>

};

90 declare function local:nameAndUrl($wine){
    element a {
        attribute href{concat($GoogleStart,encode-for-uri($wine/@name))},
        data($wine/@name)
    }
95 };

    declare function local:comment($comment){
        data($comment/../@code),' : ',local:bold($comment),<br/>
    };
100 declare function local:bold($mixed-content){
    for $child in $mixed-content/node()
        return if ($child instance of element() and local-name($child)="bold")
            then element b{data($child)}
105         else $child
};

    (: change the namespace of a subtree starting at an element
    adapted from Priscilla Walmsley, XQuery, O'Reilly, p. 258 :)
110 declare function local:change-element-ns-deep($element,$new-ns){
    element {QName($new-ns,local-name($element))} {
        $element/@*,
        for $child in $element/node()
            return if ($child instance of element())
115                 then local:change-element-ns-deep($child,$new-ns)
                    else $child
    }
};

120 (: put the whole tree in the XHTML namespace :)

```

```

local:change-element-ns-deep(
  local:root(/cellar-book),
  "http://www.w3.org/1999/xhtml")

```

20

- ❶ Parameter giving the start of the URL allowing a search for a given wine on Google.
- ❷ Function for the root node that defines the skeleton of the HTML page: a head with a title indicating the name of the owner and a body to be filled by the call to the `cellar-book` function.
- ❸ Global header with the name of the owner followed by a table giving more information about the owner and the location of the cellar.
- ❹ Loops over all child nodes of the address, except the first one, the name of the owner. After the text content of each node is rendered using the `data` function, a line break is forced with an HTML `br` element.
- ❺ Loops over all child nodes of the location. After the text content of each node is written using the `data` function, a line break is forced with an HTML `br` element.
- ❻ Calls the function to output the content of the cellar.
- ❼ The content of the cellar is given as a table with a header. It is followed by a series of lines corresponding to each wine. Finally a line holding the estimated value of the whole cellar is created.
- ❽ Calls the `wine` function for each wine but in increasing order of the `code` attribute.
- ❾ The total value of the cellar is computed using two nested `for` expressions that loop over all wines of the cellar and the catalog and return the sum of the value of each wine. The value of a wine is given by the number of bottles (`quantity`) multiplied by the price of the wine in the catalog having the same `code` attribute as this wine.
- ❿ The total number of bottles is the sum of the values of all `quantity` elements.
- ⓫ Comments appear at the bottom of the table if at least one wine has a comment. They are also given in ascending order of `code` attribute.
- ⓬ Outputs a line of an HTML table for a given wine.
- ⓭ Defines a local variable for the `code` attribute. It will be useful to differentiate it from the `code` attribute of the catalog used in the expression on line 80-❹.
- ⓮ Outputs the name of the wine and its URL using the function defined on line 90-❶.
- ⓯ Outputs a string of `*` corresponding to the number of stars for this wine.
- ⓰ Produces the name of the wine as the anchor text for an HTML link to Google. It features the appropriate wine name in order to get further information about it.
- ⓱ A comment is preceded with the value of the surrounding `code` attribute and followed by a line break.
- ⓲ A `cat:bold` element is transformed into an HTML `b` element.
- ⓳ In order to produce a legal XHTML document, all elements must be in an appropriate namespace. Because in XQuery the default global default namespace declaration also applies to the XPath expressions on the input tree, it is not possible to query a tree in the default namespace and have the output tree constructor in another. The output is first created by the transformation in the empty namespace. This result is then copied but with nodes into the XHTML namespace.
- ⓴ Calls the transformation on the `cellar-book` element and copies the result into the XHTML namespace.

6.1.3. Bulleted Lists

As we did in the previous chapter, we now describe a query that can be applied to any XML file to show its indentation structure by means of nested HTML unnumbered lists.

To transform the cellar book (Example 2.2) into the HTML code of Example 5.6 (rendered in Figure 5.3), we can use the code given in Example 6.3 which features four main functions:

- for the root element (line 20-③) the function produces the overall structure of the HTML file with its head and body elements. The enclosed expression for processing subelements (line 27-⑤) is called within an unnumbered list delimited by `ul` tags.
- for an attribute (line 33-⑥), the function returns a space, the name of the attribute followed by an equal sign and its value within double quotes.
- for an element (line 37-⑦), the function produces the name of the element returned by the function `local-name()` in bold (line 39-⑧) followed by its attributes. If the element is a node without any children (it is a text node in this case) then it is output with only its content, otherwise a new unnumbered list is started and template matching is applied on child nodes.
- a text node (line 54-⑩) is inserted into an `li` element

Example 6.3. [compactHTML.xq] XQuery script to produce a bulleted outline (Example 5.6) from the cellar book (Example 2.2). Compare this with Example 5.7.

```

1 (: to produce legal and validable XHTML ... :)
  declare option saxon:output "method=xml";                               ①
  declare option saxon:output "doctype-public=-//W3C//DTD XHTML 1.0 Strict//EN";
  declare option saxon:output "doctype-system=http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict";
5 declare option saxon:output "indent=yes";

  (: remove all whitespace nodes from elements of the input tree :)
  declare function local:strip-space($element){                          ②
    element {local-name($element)} {
10     $element/@*,
      for $child in $element/node()
      return if ($child instance of text() and normalize-space($child)='')
              then ()
              else if ($child instance of element())
15                 then local:strip-space($child)
                 else $child
    }
  };

20 declare function local:root($root,$suri){                             ③
    <html>
      <head>
        <title>HTML compaction of "{replace($suri,'.*/(.*)',''$1')}"</title> ④
      </head>
25     <body>
        <ul>
          {local:element($root)}                                         ⑤
        </ul>
      </body>
30   </html>
  };

  declare function local:attribute($attribute){                          ⑥

```

```

concat(" ",local-name($attribute),'="',$attribute,'"')
35 };

declare function local:element($element){
    <li>
        <b>{local-name($element)}</b>
40     {for $attr in $element/@*
        return local:attribute($attr),
        if (count($element/*)=0) (: single text node ?:)
        then concat(" ",data($element))
        else <ul> {
45         for $child in $element/(*|text()) (: possible mixed node :)
        return if($child instance of element()
            then local:element($child)
            else local:text($child)
        } </ul>
50     }
    </li>
};

declare function local:text($text){
55     <li>{$text}</li>
};

(: change the namespace of a subtree starting at an element
   adapted from Priscilla Walmsley, XQuery, O'Reilly, p. 258 :)
60 declare function local:change-element-ns-deep($element,$new-ns){
    element {QName($new-ns,local-name($element))} {
        $element/@*,
        for $child in $element/node()
        return if ($child instance of element()
65         then local:change-element-ns-deep($child,$new-ns)
        else $child
    }
};

70 local:change-element-ns-deep( (: put the whole tree in the XHTML namespace :)
    local:root(local:strip-space(/*),document-uri()),
    "http://www.w3.org/1999/xhtml")

```

- ❶ Declarations telling the SAXON transformation engine to serialize with the appropriate headers to produce valid XHTML.
- ❷ Function to remove all whitespace-only nodes in an element because we are only interested in the content of the input document, not its structure.
- ❸ Produces the XHTML skeleton for the document.
- ❹ Outputs the title of the HTML page. Because the \$uri parameter contains the full URI of the current document, the `replace` function uses a regular expression to keep only the part after the last slash.
- ❺ Enclosed expression for starting the processing at the root element node.
- ❻ An attribute is output as a space, the name of the attribute and its value between double quotes after an equal sign. This sequence of nodes is within a call to `concat` so that no spurious spaces are inserted in the output.

- ⑦ An element node is a *list-item*.
- ⑧ The name of the element in bold, followed by the attributes using the function defined on line 33-⑧.
- ⑨ If there are no children elements, i.e. it is a single text node, the textual content is inserted directly without adding a new list level.
- ⑩ If there are children nodes, elements are processed recursively and text nodes (in the case of mixed content) are processed with the `text` function.
- ⑪ If there are no children nodes, it is then a text node that is copied to the output.
- ⑫ In order to produce a legal XHTML document, all elements must be in an appropriate namespace. Because in XQuery the default global default namespace declaration also applies to the XPath expressions on the input tree, it is not possible to query a tree in the default namespace and have the output tree constructor into another. The output is first created by the transformation in the empty namespace. This result is then copied but with nodes into the XHTML namespace.
- ⑬ Calls the transformation on the `cellar-book` element and copies the result into the XHTML namespace.

6.2. Transformation into a Compact Textual Form with XQuery

We will now use XQuery query (shown in Example 6.4) to produce the compact form of an XML file. The output of this transformation will only be a stream of *plain* characters without any tags. This shows that XQuery can be used to produce a text file.

The algorithm given in Example 6.4 follows the same pattern as the one explained in Section 6.1.3. We recursively follow the structure of the tree and output a corresponding stream of characters. Sequences of character elements are always embedded within a `concat` function so that spurious spaces are not added between them. The `element` function starting on line 29-⑥ is applied to all element nodes: we output the name of the element and then output its attributes and children, with an indentation corresponding to the number of characters in the name of the parent element. The root node has an indentation of 0. Because we want an output with few blank lines, we only change line after having written the first attribute or child element. This is implemented within the loop on each child and attribute. Because XQuery explicitly *pulls* the input tokens from the input, this test can be done at only one place in the program contrarily to the XSLT program in Example 5.9 which uses a push approach and thus must test its position in each case. We could also have used the pull approach in XSLT, but the push approach is more *natural* in XSLT because of the implicit application of templates according to the structure of the document.

For each child node and attribute returned in document order, we call the appropriate function depending on the type of node.

On line 21-④, we define the function to output the value of an attribute, which is simply the name of the attribute preceded by an @ and followed by its value in square brackets.

On line 25-⑤, we output the value of the current node with all extraneous space removed using the `normalize-space` function. This removes all whitespace at the start and at the end of the value and leaves only one space between *non-space* characters. Because the `strip-space` function (line 5-②) will have been called on the input document before processing, nodes containing only newlines and spaces will not appear in the source tree.

To output a given number of spaces, we have defined a function called `nl-and-indent` (line 17-③) taking one parameter used to create a sequence of spaces, which are joined into one string preceded by a newline.

Example 6.4. [compact.xq]: Stylesheet used to transform the cellar book instance document (Example 2.2) into Example 5.8. Compare this with Example 5.9.

```

1 declare option saxon:output "method=text";                                ❶
  declare option saxon:output "omit-xml-declaration=yes";

  (: remove all whitespace nodes from elements of the input tree :)
5 declare function local:strip-space($element){                             ❷
    element {local-name($element)} {
      $element/@*,
      for $child in $element/node()
      return if ($child instance of text() and normalize-space($child)='')
10         then ()
        else if ($child instance of element())
            then local:strip-space($child)
            else $child
    }
15 };

  declare function local:nl-and-indent($nb){                               ❸
    concat('&#xA;',string-join(for $i in 1 to $nb return ' ',''))
  };
20

  declare function local:attribute($attribute){                           ❹
    concat('@',local-name($attribute),'[',data($attribute),']')
  };

25 declare function local:text($text){                                     ❺
    normalize-space($text)
  };

  declare function local:element($elem,$indent){                          ❻
30   let $newName := local-name($elem),
      $newIndent := $indent+string-length($newName)+1
    return
      concat(
        $newName,"[",
35       string-join(
          for $child at $pos in $elem/(@*|*|text())
          return (
            if($pos>1)
            then local:nl-and-indent($newIndent)
40         else (),
          typeswitch ($child)                                             ❼
            case attribute() return local:attribute($child)
            case text()      return local:text($child)
            case element()   return local:element($child,$newIndent)
45         default return ()
        )
      )
  }

```

```

        ), "" ),
        "]"
    )
};
50 local:element(local:strip-space(/*),0)

```

- ❶ Indicates that the output will be plain text, thus we do not want an XML declaration to be emitted.
- ❷ Function to remove all whitespace-only nodes in an element because we are only interested in the content of the input document, not its structure.
- ❸ Function for outputting a number of spaces given by the `nb` parameter preceded by a new line (indicated by a character entity).
- ❹ For an attribute, outputs the name of the attribute preceded by an `@` followed by its value enclosed in square brackets.
- ❺ For a text node, outputs its normalized value, i.e. removes the surrounding spaces and line breaks.
- ❻ For an element, first outputs the element name followed by an opening bracket. It then recursively calls the compaction of attributes, children and text nodes. Finally a closing bracket is output. If there are any attributes and this element is not the first one, indent the following line.
- ❼ Depending on the type of the child node, calls the appropriate function. in the case of an element node, does a recursive call to `element` but with an updated `indent` parameter.
- ❽ At the root, starts to apply the compaction algorithm with an indentation of 0.

6.3. Querying an instance file

The main motivation for using XQuery is for querying XML databases. Most commercial relational database systems now offer a way to keep and select information in XML but some XML native databases have been developed. `eXist-db` [65] is an open source database management system that stores XML data according to the XML data model and features efficient index-based XQuery processing.

XQuery can be called from the command line with the Saxon jar library [83]. For example, calling the query Example 6.1 and giving `white` as value of the `$color` parameter can be done as follows:

```
java -cp /path/to/the/SAXON.jar net.sf.saxon.Query -s WineCatalog.xml \
-q WineCatalog.xq color=white > whites.html
```

6.4. Additional Information on XQuery

The official information on XQuery [10] is comprehensive and detailed, so much so in fact that it counts more than 170 printed pages. A more tutorial approach can be found in the XQuery book by Wamlsley [61] which gives good examples of use and tricks of the trade. Another excellent source of examples are given in the XQuery Cookbook [54]. It also gives examples of use of for the `eXist` database and how XQuery can be used with programming languages and environments.

Chapter 7. RDF : Resource Description Framework

In the previous chapters, we have seen how to label information using XML tags so that it can be processed with diverse technologies. Although, we tried to give mnemonic names to element tags, we never assigned any formal meaning to them. We focused on the *syntactic* checking of the tag names and their proper nesting according to a schema. In order that the information on the web be more easily shared and manipulated by machines or *automated intelligent agents*, much research activity has been performed in the area of the **Semantic Web** [8] to make available for applications not only the information textually present in documents but also its intended meaning.

On the Semantic Web, computers, not only humans should do the browsing in order to search for knowledge, to process it and to take action. Now that the current (*syntactic*) web is a well-accepted distributed **presentation** platform, the semantic web aims for a distributed **knowledge** platform. Although this is a laudable goal, it is much easier said than done. Progress in this area has been relatively slow compared to the expectations of the original Semantic Web designers. But ideas and technologies have emerged and started being used in many contexts.

In this chapter, we present a Semantic Web technology called RDF which stands for Resource Description Framework. RDF is a way of identifying relations between things on the web; it is very flexible and scalable but as it is quite loose, it is limited in its inferencing possibilities. XML and RDF are at the core of the Semantic Web as can be seen in Figure 7.1. RDF can be described using an XML syntax although its principles are independent from this notation. RDF should be primarily considered as a triple-based data model to define graphs of relations between *things*.

We will first present the data model using a non-XML syntax for which an XML serialization has been defined. For humans, the non-XML syntax is easier to read, to write and understand but the uniformity, ease of parsing and standardisation of the XML syntax is a great asset for computer-based applications.

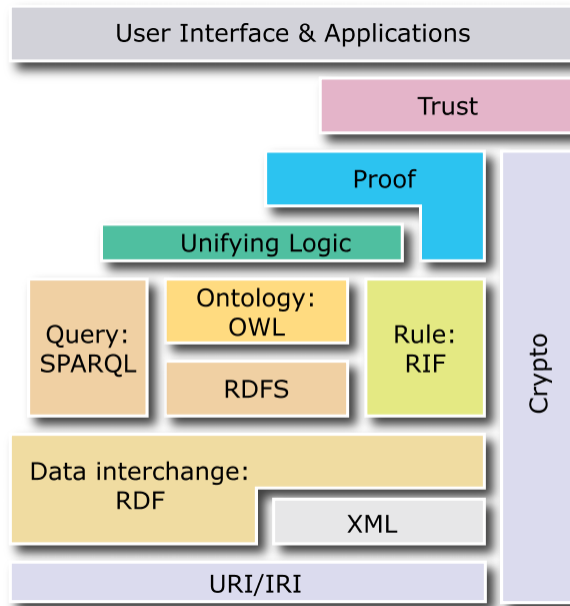
RDF [78] was developed as a simple and extensible data model for annotating web content. Its goal was to enable *Anybody to say Anything about Any topic*, sometimes called the *AAA Slogan*. It is a tradeoff between ease of expression and strictness of organization that allows the combination of information from diverse sources anywhere on the web or even elsewhere. RDF is a means of associating property-value pairs with resources on the web. It is more oriented towards meta-information about web pages than providing information itself although its generality allows the definition of anything.

In order to associate properties and values to entities, it is important to be able to identify them uniquely and universally. RDF uses the notion of URI which stands for *Uniform Resource Identifier*. An URI can identify anything (a *resource* in the RDF jargon) either

a web page	<code>http://www.iro.umontreal.ca/~lapalme/ForestInsteadOfTheTrees/HIHL/index.html</code>
a web site	<code>http://rali.iro.umontreal.ca</code>
a person	<code>http://www.iro.umontreal.ca/lapalme</code>
an abstract concept	<code>http://www.iro.umontreal.ca/lapalme/MyViewOnXML</code>

Figure 7.1. Semantic Stack

"Layer cake" diagram (2007 version) taken from <http://www.w3.org/sw>



An URI is in a sense arbitrary but, like the namespaces we described in Section 2.1, they often look like URLs whose start usually give an official name or address of a person or organization. An URI **is not necessarily an existing web document**. Although there is no formal check on this, it is expected that a person or an organization defining an URI for their own purposes will start with an address that they *own* in a certain way unless an existing resource is referred to. It usually consists of the following parts:

- scheme a type of address followed by a colon, most often `http:`, but it could be `mailto:`, `isbn:` or `tel:` among others.
- authority the name of a domain in the internet (e.g. `www.iro.umontreal.ca`) preceded by `//`
- path usually your name or the name of a project possibly followed by the specifics of this application (e.g. `lapalme/CellarBook`);
- fragment starting with hash (`#`), a specific term for the name given to the concept to identify (e.g. `#wine`).

These would form the following URI:

```
http://www.iro.umontreal.ca/lapalme/CellarBook#wine
```

Although we have argued previously that URIs are arbitrary, the goal of the Semantic Web is to share information with others. Naming conventions are thus essential to ensure that the same concepts or information are all named the same. Also a given name should not reference different things depending on the context of use because URI are *universal*.

The largest effort to distribute and link structured information across the web is a project called *Linked Data* [30]. In this context, a naming convention patterned after the *Cool URI* [41] proposition of the W3C has been defined to ease the sharing of information. Here are a few guidelines for creating one's own URIs so that others can easily make sense of them and combine them in their applications.

There are two requirements for a *cool uri*:

- **Be on the web:** given only a URI, machines and people should be able to retrieve a description about the resource identified by the URI from the Web.
- **Be unambiguous:** there should be no confusion between identifiers for Web documents and identifiers for other resources. For example, there should be a different URI for referencing the author of this document and his web page. URIs are meant to identify only one of them, so one URI can't stand for both a Web document and a real-world object.

These two contradictory requirements can be met by taking advantage of the HTTP protocol which allows (at least) two *tricks* : *hash URIs* and *303 URIs*.

Hash URIs are URLs containing a fragment part i.e. an identifier following a hash sign (#) thus their name. Hash URIs are used for non-document resources. When a client sends a request to retrieve a hash URI from a server, the HTTP protocol requires that the fragment part be stripped off before requesting the URI from the server. Thus a URI with a hash cannot be retrieved directly and does not necessarily identify a Web document but it can be used to identify a non-document resource without creating ambiguity. This scheme is often used when many concepts are defined in the same HTML or RDF document. Referencing the URI thus retrieves a human or machine readable document giving access to all these concepts with a single access.

303 URIs are mostly used when the RDF and HTML representations of the resource are two different documents. In this case, the Web server corresponding to a given URI can be configured to redirect the query to another URL depending on the information given by the caller in the header of the request: e.g. it could answer with an HTML document to a browser but with an RDF document if the caller is an application. By looking at the code of the response header (200 for a normal document and 303 for a redirection), the application can distinguish the two types of response.

Within the Linked Data community, it is a best practice to distinguish three types of URI for non-information resources: the resource itself, a web page for human readers or a RDF/XML file for machines.

For example in DBpedia, the town of Montréal in Canada is associated with three URIs:

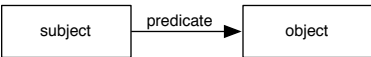
- <http://dbpedia.org/resource/Montreal> : Montréal as a town;
- <http://dbpedia.org/page/Montreal>: human oriented description in HTML about Montréal;
- <http://dbpedia.org/data/Montreal>: machine readable information in RDF/XML about Montréal

The choice between these two mechanisms depends on the application; section 4.4 of [41] present some basic rules but it concludes by “If in doubt, follow your nose”! But care has to be given for the initial choice because in the words of Tim Berners-Lee “Cool URIs don't change”!

7.1. Triples in RDF/XML

The data model of RDF is deceptively simple: it merely defines a directed binary relation between two resources: the binary relation is called a *predicate* or a *property*; the two resources are *subject* and *object*. These three elements form a *triple* and they are all identified by URIs. An object can be a constant data value instead of a URI but the subject and predicate must be URIs. Table 7.1 shows different equivalent notations for an RDF triple. The RDF/XML is the most widely used for storing and exchanging RDF information. The equivalent triple (or its more recent variant *Turtle* [88]) is easier for humans to type and to understand. The graphical form can also be useful to grasp relations between many resources, but it can become quite hairy when there are many resources and relations. A very useful tool for converting between different RDF notations is the *RDF validator* [79] which not only validates the input but can produce the corresponding graph or show the equivalent triples. *Twinkle* [89] provides a GUI interface to query RDF data, but it can also parse and produce the output of the queries in different notations of RDF.

Table 7.1. RDF Triples in different forms, all of them equivalent

Graphical form	
Triple	subject predicate object
Relational form	predicate(subject, object)
RDF/XML	<pre><rdf:Description rdf:about="subject"> <ex:predicate> <rdf:Description rdf:about="object"/> </ex:predicate> </rdf:Description></pre>
Turtle	subject ex:predicate object.

Going from top to bottom, we see: the graphical notation in which the subject and object are linked by a labelled arrow; the triple format in which each component is a URI or data value in the case of an object; the relational form that would be the equivalent in Prolog; the RDF/XML exchange format and the Turtle format which, in the case of a single triple, only differs from the triple notation by its use of namespace prefix in URIs.

Information is represented by a set of RDF triples, often called a *model*. As it is a set, there is no ordering, nor repetition between triples within a model. Information is encoded as a **conjunction** of triples. Negation and disjunction cannot be expressed in RDF.

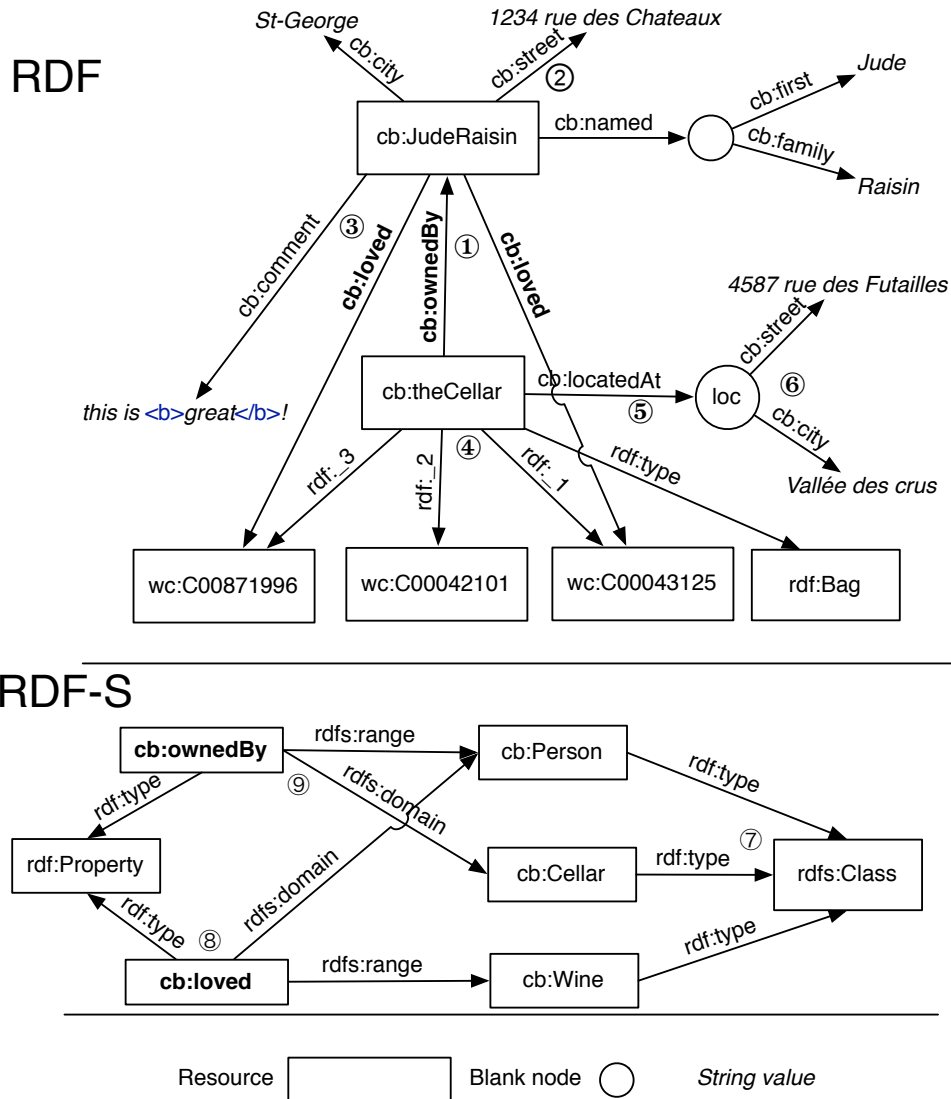
All these RDF notations are equivalent, in fact the first thing an RDF parser does when it reads a file, is to transform the information in terms of triples, removing duplicates. The output files are produced by serializing the triples in the same or in another format.

Given the emphasis on XML in this document, we will first focus on the RDF/XML notation. But first, we will explain the concepts with a small set of relations given in the top part of Figure 7.2, the bottom part will be explained in the next section. Example 7.1 gives the corresponding RDF/XML version. As RDF predicates can link any subject and object, an arbitrary network of information can be built which is not necessarily limited to a tree like in XML

As shown in the fourth line of Table 7.1, a triple in RDF/XML, is formed by a `rdf:Description` element whose content is an element named by the URI of the predicate having the object as content (see line 13-5 of Example 7.1). By embedding a predicate and its object within a subject, it is easy to regroup triples that share a subject by writing all predicates and their object within the subject element.

Figure 7.2. Selected information from the cellar-book

Graphical representation of triples. The circled numbers are not part of the graphical representation, they are only given here for reference purposes. They correspond to the numbers given in XML comments besides each `rdf:Description` in Example 7.1 and Example 7.2. The top part of this diagram deals with the RDF while the bottom part deals with the RDF Schema (RDFS) explained in the next section.



A triple can only indicate a binary relation. To indicate an n-ary relation between a subject and n objects, we make use of a blank node (indicated by a circle in Figure 7.2) which is a node with a local URI different from any other one in another file. We first make a relation between the subject and the blank node and then other relations between the blank node and the other objects. For example, to indicate that the cellar-book is located at a given street in a certain city, we first link the cellar-book with a blank node which is then linked to the name of the street and to the name of the city (see the arrow marked by a circled 5 in Figure 7.2).

Components of a triple are URIs which are usually quite long strings many of them varying only at the end within the same file, so it is quite convenient to abbreviate them using entities for the constant part. As the predicates are elements, the constant part can be given as the namespace of the element. A practical way of achieving this is to define an entity and the corresponding namespace prefix with the same letter. As the entity is defined for the whole file, we can also use it for defining the namespace prefix thus ensuring that they always correspond. We use this approach in Example 7.1 at line 3-② and line 7-③. Another way of abbreviating URIs is to make them relative to another one using `xml:base`, see line 11-④ of Example 7.1.

An object can either be a URI or a string value. In the latter case, the string is given as the content of the predicate element. The predicate and its string value can also appear as an attribute with its value of the `rdf:Description` element of the subject (see line 20-⑥ of Example 7.1).

Example 7.1. [CBWC-RDF-S.rdf] Subject, Predicate and Object triples for the cellar book in RDF/XML.

Selected information from Example 2.2 coded in RDF/XML. The information was selected to illustrate different features of RDF/XML.

```
1 <?xml version="1.0" encoding="UTF-8"?>
  <?oxygen RNGSchema="rdfxml.rnc" type="compact"?> ①
  <!DOCTYPE RDF [ ②
    <!ENTITY wc "http://www.iro.umontreal.ca/lapalme/WineCatalog#" >
5    <!ENTITY cb "http://www.iro.umontreal.ca/lapalme/CellarBook" >
  ]>
  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" ③
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:wc="&wc;"
10    xmlns:cb="&cb;#"
    xml:base="&cb;"> ④

    <rdf:Description ⑤
      rdf:about="http://www.iro.umontreal.ca/lapalme/CellarBook#theCellar">
15    <cb:ownedBy>
      <rdf:Description rdf:about="&cb;#JudeRaisin"/>
    </cb:ownedBy>
    </rdf:Description>

20    <rdf:Description ⑥
      rdf:about="http://www.iro.umontreal.ca/lapalme/CellarBook#JudeRaisin"
      cb:street="1234 rue des Châteaux" cb:city="St-George">
      <cb:named rdf:parseType="Resource"> <!-- 2 -->
        <cb:first>Jude</cb:first>
25        <cb:family>Raisin</cb:family>
      </cb:named>
    </rdf:Description>

30    <rdf:Description rdf:about="JudeRaisin"><!-- 3 --> ⑦
      <cb:loved rdf:resource="&wc;C00043125"/>
      <cb:loved rdf:resource="&wc;C00871996"/>
      <cb:comment rdf:parseType="Literal">this is <b>great</b>!</cb:comment>
    </rdf:Description>
```



```

35   <rdf:Bag rdf:ID="theCellar">                                <!-- 4 -->           ⑧
      <rdf:li rdf:resource="&wc;C00043125" />
      <rdf:li rdf:resource="&wc;C00042101" />
      <rdf:li rdf:resource="&wc;C00871996" />
    </rdf:Bag>

40   <rdf:Description rdf:about="#theCellar"><!-- 5 -->         ⑨
      <cb:locatedAt rdf:nodeID="loc" />
    </rdf:Description>

45   <rdf:Description rdf:nodeID="loc">                       <!-- 6 -->         ⑩
      <cb:street>4587 des Futailles</cb:street>
      <cb:city>Vallée des crus</cb:city>
    </rdf:Description>

50   <!-- Schema definition -->                                ⑪

      <!-- Classes -->
      <rdfs:Class rdf:ID="Person" />                           <!-- 7 -->         ⑫
55   <rdfs:Class rdf:ID="Cellar" />
      <rdfs:Class rdf:ID="Wine" />

      <!-- Properties -->
      <rdf:Property rdf:ID="loved">                             <!-- 8 -->         ⑬
60   <rdfs:domain rdf:resource="#Person" />
      <rdfs:range rdf:resource="#Wine" />
    </rdf:Property>

      <rdf:Property rdf:ID="ownedBy">                          <!-- 9 -->         ⑭
65   <rdfs:domain rdf:resource="#Cellar" />
      <rdfs:range rdf:resource="#Person" />
    </rdf:Property>

</rdf:RDF>

```

- ① XML validation can be done using the Relax-NG schema given in appendix A of the RDF definition [78]. Note that this validation only validates the form of the XML file and is different from the schema notion used in RDFS explained in the next section.
- ② Defines two entities to simplify the writing of an URI. `wc` ends with a `#` as it will be used to separate fragments within the URI. `cb` will be used as an URI for a base so it does not end with a `#`.
- ③ Starts an RDF model by setting the appropriate namespaces. `rdf` must be this one, but the others are application dependant. Here we define two namespaces: `wc` for the wine-catalog and `cb` for the cellar-book. Using the entities defined in line 3-②, it guarantees that predicates using the namespaces and the URI for the identification of the resources are consistent.
- ④ Defines the base for relative URI in this document indicated by `rdf:ID`, such as for line 35-⑧. When no `xml:base` is specified, relative URI use the current document as base. This is useful to shorten the URIs in a RDF document. Note that the base is not used for predicates which must use the namespace prefix.

- ⑤ The cellar-book is owned by the resource identified by `JudeRaisin`. The straightforward way for noting a triple: both the subject and object are designated by their URI in a `rdf:Description` element. For illustration purpose, we use here the full URI for the subject and the abbreviated form with an entity for the object. Both notations refer to the same resource as would be one using `rdf:ID` (see line 29-⑦). The predicate is indicated by an element named by its URI using the namespace prefix.
- ⑥ Description of `#JudeRaisin` with a complex node and attributes. Predicates with a string value can be inserted as attributes of the subject. A predicate with complex content will define a blank node which will act as subject of the content. This fact must be indicated by giving `Resource` as value of the `rdf:parseType` attribute.
- ⑦ `#JudeRaisin` loved two wines of the catalog and expressed this fact by a comment. Predicates can link any two nodes in the graph. The global model is thus an arbitrary graph and not necessarily a tree. To embed XML content as an object, we must specify `Literal` as value of the `rdf:parseType` attribute.
- ⑧ The cellar-book, referred by the id `CellarBook` (corresponding to the full URI given in line 13-⑤) contains three wines not further defined in this file. We use a `rdf:Bag` here instead of a `rdf:Description` to indicate a *container*. In a *bag*, the order of the elements, noted by `rdf:li` is not important. Should the order be important, we could have used a `rdf:Seq`. `rdf:Alt` would indicate an alternative between many cases. The graph representation of a container declaration is a link between the resource and the `rdf:bag` URI; the `rdf:li` elements are represented by links denoted by `rdf:_n` where *n* is a number corresponding to the order in which the corresponding `rdf:li` element appeared in the document.
- ⑨ The cellar-book has complex information, so it is regrouped under a blank node referred to by a `nodeID` which will be defined at line 45-⑩. Contrarily to a node referred by a `rdf:ID`, the id of a blank node, referred by `rdf:nodeId` is local to this file.
- ⑩ Regrouping the information about the location within a blank node. The string information of the object is given as the value of the predicate element.
- ⑪ Start of the RDFS part that will be further explained in the next section.
- ⑫ Definition of three classes identified with IDs within the namespace identified by the `cb` entity because of the `xml:base` declaration. A class declaration is represented in RDF by a `rdf:type` link between the class name and the predefined `rdfs:Class` class which is the class of all classes as shown by lines numbered 7 in Figure 7.2.
- ⑬ Defines the `loved` property linking an instance of a `Person` class as its domain (i.e. the start of the arrow) with an instance of a `Wine` class, the range (i.e. the tip of the arrow). These classes were identified in line 54-⑭. A property definition also adds a `rdf:type` link between the name of the property and the predefined `rdf:Property` (yes, `Property` is in the `rdf` namespace and not `rdfs`).
- ⑭ Defines a `ownedBy` property linking an instance of a `Cellar` class with an instance of a `Person` class.

Although RDF/XML is a convenient triple notation for computers, it is not very user-friendly. So alternatives have been developed, one of the most widespread being the Terse RDF Triple Language more commonly called *Turtle* [88]. The main rules for writing Turtle triplets are the following:

- A simple triple is a sequence of three *terms* (subject, predicate, object) separated by spaces and ending with a period.
- There are three types of *terms*:

URI	written between <code><</code> and <code>></code> ; an URI can also be written as a <code>QNAME</code> i.e. an identifier preceded by a namespace prefix and a colon. A namespace prefix is defined with a <code>@prefix</code>
-----	---

definition. The full URI is obtained by concatenating the content of the namespace prefix with the rest of the r-uri; A namespace prefix can be empty but the : still appears.

literal written between single or double-quotes, possibly followed by a type definition marked by ^^ followed by a type name, most often of the form xsd:... for standard XML Schema types.

blank node marked by _:nodeId, a [] or a parenthesized group (explained below)

- a namespace prefix p is defined by @prefix p: URI.
- a comma repeats the subject and the predicate that differ only by their object, e.g. ex:a ex:b ex:c, ex:d. is the same as ex:a ex:b ex:c. ex:a ex:b ex:d.
- a semi-colon repeats the subject of triples that differ by their predicate and object, e.g. ex:a ex:b ex:c; ex:d ex:e. is the same as ex:a ex:b ex:c. ex:a ex:d ex:e.

A blank node of the form _:nodeId can also be noted as [] when the nodeId is not used elsewhere. A blank node as subject such as [] ex:p "o" can be noted as [ex:p "o"]. This type of bracketing can be used in either subject or object position and combined with commas and semi-colons. Table 7.2 compares the bracket notation with the corresponding triples using blank nodes.

Table 7.2. Comparison of Turtle brackets with triples

[] ex:p "o" .	_:b0 ex:p "o" .
[ex:p "o"] .	_:b1 ex:p "o" .
[ex:p "o"] ex:q ex:o2 .	_:b2 ex:p "o" . _:b2 ex:q ex:o2 .
ex:s ex:p [ex:p1 ex:o] .	ex:s ex:p _:b3 . _:b3 ex:p1 ex:o .
[ex:p ex:o ; ex:p1 "o1"]	_:b4 ex:p ex:o . _:b4 ex:p1 "o1" .

Comparison between the bracketed notation for blank nodes and the corresponding triples.

Jena [69] provides libraries and a command line tool `rdfcopy` for parsing and writing RDF triples in different notations, thus providing a simple conversion tool between these notations.

Example 7.2. [CBWC-RDF-S.ttl] The Turtle version of Example 7.1

An easier to type and to read version of the RDF information described in Example 7.1. As before, this section only deals with the RDF part and not the RDFS part starting at line 22-7. In Turtle, comments start with # and span for the rest of the line. In this example, the numbers in comments correspond to the lines with the same numbers within XML comments in Example 7.1.

```

1 @prefix rdf:      <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .      ❶
  @prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
  @prefix cb:     <http://www.iro.umontreal.ca/lapalme/CellarBook#> .
  @prefix wc:     <http://www.iro.umontreal.ca/lapalme/WineCatalog#> .
5
  cb:theCellar
    cb:ownedBy cb:JudeRaisin ; # 1
    a         rdf:Bag ; # 4
    rdf:_1   wc:C00043125 ;
    
```

```

10      rdf:_2  wc:C00042101 ;
      rdf:_3  wc:C00871996 ;
      cb:locatedAt [ cb:city "Vallée des crus" ;          # 5,6
                    cb:street "4587 des Futailles" ] .

15 cb:JudeRaisin cb:city "St-George" ;                # 2      ④
      cb:named [ cb:family "Raisin" ;                ⑤
                 cb:first "Jude" ] ;
      cb:loved wc:C00043125 , wc:C00871996 ;          # 3      ⑥
      cb:comment "this is <b>great</b>!"^^rdf:XMLLiteral ;
20      cb:street "1234 rue des Châteaux" .

      # Schema part in RDFS                                ⑦

      cb:Person a      rdfs:Class .                    # 7      ⑧
25 cb:Cellar a      rdfs:Class .
      cb:Bottle a     rdfs:Class .

      cb:loved
          a      rdf:Property ;                          # 8      ⑨
30      rdfs:domain cb:Person ;
          rdfs:range cb:Wine .

      cb:ownedBy
          a      rdf:Property ;
35      rdfs:domain cb:Cellar ;
          rdfs:range cb:Person .

```

- ① Definition of namespace prefixes. As prefixes can also be used for subjects and objects of triples, which is not the case in RDF/XML, base definition is less useful, but still possible.
- ② Six predicates sharing `cb:CellarBook` as subject.
- ③ The notation for a container in Turtle corresponds to the graphical representation: here, the bag is represented by a link to the predefined `rdf:Bag` type (`a` is an abbreviation for `rdf:type`) followed by links of the form `rdf:_n` where `n` is a number indicating the order in which the `rdf:li` appears in the document. These triples correspond with the arrows with a circled 4 in Figure 7.2.
- ④ Four predicates having `cb:JudeRaisin` as subject are grouped, their predicate and object being separated by semicolons.
- ⑤ A blank node is used as object of `cb:named` and subject of both `cb:family` and `cb:first`, this explains the semi-colon within the square brackets.
- ⑥ Two objects in namespaces `wc` also share both their subject `cb:JudeRaisin` and their predicate `cb:loved`, so they are separated by commas.
- ⑦ Start of the RDFS part further described in the next section.
- ⑧ Definition of three classes in `cb` namespace. A class declaration is represented in RDF by a link (which correspond to `rdf:type`) between the class name and the predefined `rdfs:Class` class which is the class of all classes as shown by lines numbered 7 in Figure 7.2.
- ⑨ Defines the `loved` property linking an instance of a `Person` class as its domain (i.e. the start of the arrow) with an instance of a `Wine` class, the range (i.e. the tip of the arrow). These classes were identified in line 24-⑧. A property definition also adds a link between the name of the property and the predefined `rdf:Property`.

- ⑩ Defines a `ownedBy` property linking an instance of a `Cellar` class with an instance of a `Person` class.

7.2. RDF Schema

RDF describes simple statements about individuals and relations between them using some basic types without any structure. This allows great freedom in the creation of new individuals but it would often be useful to rely on a better organisation, so that relations can be typed and thus allow some inferences from a given set of facts:

The relations (or properties):

- **loved** links a *person* with a *wine*
- **owned by** links a *cellar* with a *person*

can seem obvious to a human but, for a machine, knowledge about the vocabulary must be made explicit. RDFS is a language designed for such cases and it was given a semantics that allows to deduce implicit knowledge (i.e. new RDF triples) from existing triples.

RDFS is a special case of RDF, every RDFS document is also an RDF document. RDFS allows generic constructs to define a particular vocabulary. As shown in Example 7.1, RDFS can be written with the fact that it describes in the same RDF document. The resulting document thus carries its own semantics. RDFS is a light ontology language to define classes and instances.

RDFS words are defined in a standard namespace `http://www.w3.org/2000/01/rdf-schema#` (see line 7-③ of Example 7.1 and line 1-① of Example 7.2)

A word of caution : although RDFS defines a schema, it does not enforce any constraint on the input of the information. This is in stark contrast with an XML schema whose role is to validate the content of the file. The RDF schema allows an inference system, called a *reasoner* in Semantic Web parlance, to infer (to entail in the Semantic Web jargon) new data from the ones that are given explicitly in the file. For example, from line 29-⑦ of Example 7.1, or the equivalent in line 18-⑥ of Example 7.2, where we declare that the property `cb:loved` links a person with a wine line 59-③ of Example 7.1 (line 28-⑨ of Example 7.2), the reasoner could infer that `#JudeRaisin` is an instance of the class `Person` and that `C00043125` is an instance of the class `Wine`.

Similarly, from line 59-③ of Example 7.1 (line 28-⑨ of Example 7.2), it could infer that `Cellar` is of class `CellarBook` and infer *again* that `JudeRaisin` is of class `Person`, but that fact will only be kept once in the data base of triples.

Note that nothing in RDF precludes that a resource be a member of many classes even ones that do not make sense for a human, e.g. a resource could be an instance of both a human and wine.

RDFS is thus a way of adding some semantics to an RDF file. It allows the definition of classes and properties but also of subclasses and subproperties that are not used in this simple example.

For more details on the semantics associated with RDFS, one should consult [24].

7.3. RDF queries

As RDF has an XML serialization, we might be tempted to use XSLT to query RDF content, but this is not practical because XSLT would be limited to a given form of XML serialization which is not unique. XSLT is strongly dependent on the syntactic organisation, i.e. the names and nesting of the elements of the XML document. But as we saw above, RDF is notation for a set of triples with many XML possible serializations. RDF queries should instead be based on the underlying triple-based data model and not on its XML form.

The RDF data model being triples, the W3C has proposed SPARQL¹, a query language analogous in goal with SQL in which it is possible to retrieve all triples in the model that correspond to a given pattern. SPARQL uses a notation similar to Turtle to identify triples to be extracted from the RDF database. The main difference being that the subject, the predicate or the object of a triple can be a *variable*, noted by an identifier preceded by ? or \$. Such a triple is called *triple pattern*, many triple patterns can be combined to form a *graph pattern*. For example, in Example 7.2: the triple pattern

```
cb:JudeRaisin cb:street ?address
```

matches the triple

```
cb:JudeRaisin cb:street "1234 rue des Châteaux"
```

The triple

```
cb:JudeRaisin cb:loved ?wine
```

matches two triples corresponding to the two wines.

```
?s ?p ?o
```

matches all triples. The variable name used in many triple patterns is constrained to the same value within a graph pattern.

It is thus possible to create more complex queries. For example,

```
cb:CellarBook cb:ownedBy ?owner.  
?owner cb:loved ?wine.
```

would match variables ?owner to cb:JudeRaisin and ?wine to wc:C00043125 and wc:C00871996. Should the value of the variable ?owner not be needed, then we can use and blank node as variable simplified using the bracket notation as follows

```
cb:CellarBook cb:ownedBy [cb:loved ?wine].
```

Graph patterns are used in two main types of SPARQL queries:

<pre>SELECT vars where {graph pattern}</pre>	The output of this query is a table containing the values of the selected variables from all triples that match the graph pattern.
<pre>CONSTRUCT {graph pattern₁} vars where {graph pattern₂}</pre>	The values of the variables in triples matching the second graph pattern are determined and used in building new triples corresponding to the first graph pattern. Variables are shared between both graph patterns.

¹SPARQL is a recursive acronym of SPARQL Protocol and RDF Query Language

Example 7.3. [CBWC-RDF-S.rq] SPARQL queries on Example 7.2

SPARQL queries to select triples. # is used to comment the rest of the line. As only one SELECT or CONSTRUCT query can be used at a time on RDF database, the other queries should be commented out before executing.

```

1 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>           ❶
  PREFIX cb: <http://www.iro.umontreal.ca/lapalme/CellarBook#>
  PREFIX wc: <http://www.iro.umontreal.ca/lapalme/WineCatalog#>

5 # show all triples                                           ❷
  SELECT * WHERE {?s ?p ?o}

  # show the loved wines of the CellarBook                       ❸
  SELECT * WHERE {[cb:loved ?wine]}

10 # show the street on which the owner of the Cellarbook lives
  SELECT ?street WHERE {cb:CellarBook cb:ownedBy [cb:street ?street]} ❹

  # show all wines of the cellar                                 ❺
15 SELECT ?wine WHERE {cb:CellarBook ?x ?wine FILTER regex(str(?x),"_[0-9]+$")}

  # define a new triple with an inverse relation                 ❻
  CONSTRUCT {?who cb:owns ?what} WHERE {?what cb:ownedBy ?who}

```

- ❶ Define namespace prefixes to be used in triples. The prefix declaration has slightly different syntax than the one used in Turtle (see line 1-❶ of Example 7.2)
- ❷ Show all triples. * indicates that the value of all variables used in the queries will be displayed.
- ❸ Display the loved wines from the cellar. As the name of the subject is not needed here, we used a blank node as variable as follows [] cb:loved ?wine which can be abbreviated with the bracket notation.
- ❹ Use the bracket notation to simplify getting information out of a complex graph structure.
- ❺ Display all the wines that are part of the Bag. To get them, we filter the triples by matching a regular expression on the string form of the URI. We check that it ends with an underscore followed by numbers.
- ❻ Create a new triple from an existing triple.

Twinkle [89] is a Java application that provides a graphical user interface for SPARQL queries on RDF models. These models can either be local files or any file on the internet. More details about the syntax of SPARQL can be found at [84]. A list of simple queries is shown in [55]. There is also a Java API, used by Twinkle, to make SPARQL queries on an RDF database and get the corresponding triples.

7.4. RDF versus XML

Although both RDF and XML are used for codifying information, there are some fundamental differences between them which can be summarized as given by an information sheet issued by *Semaview.com* (not in business anymore):

- XML is a tree, RDF is a set of triples;
- XML is ordered, but RDF is not ordered;

- RDF uses XML as one serialization mean for its data model built upon triples;
- RDF is easier to subset;
- RDF is easier to query using SPARQL.
- RDF allows some abstraction from the document syntax;
- RDF is more complicated and requires some planning;
- XML is syntactic, RDF is semantic;
- Not all projects need RDF.

7.5. Additional Information on RDF

This chapter only gave some of the basic information about RDF. Good books have been published in recent years. Practical views of the semantic web technologies are shown in [24] and programming aspects are dealt by [58]. More formal aspects are described clearly in [24].

There are many information sources about RDF on the web, some of them being:

<http://www.w3.org/RDF/>

The primary official source with plenty of links about tools and example applications.

<http://www.rdfabout.com>

Excellent introductory material about RDF and some data sources.

<http://jena.sourceforge.net/>

A Semantic Web Framework for Java

<http://librdf.org/>

C libraries that provide support for RDF with language Bindings in Perl, PHP, Python and Ruby

<http://dbpedia.org/>

The DBpedia project extracts various kinds of structured information from Wikipedia and combines this information into an RDF knowledge base.

Chapter 8. Document Processing by Programming in Java

In the previous sections we have described how to process XML files with XML declarative tools. But it is also possible to use standard programming languages to process XML files, thus allowing a much finer control on the output and a better integration with other types of processing. Processing XML files is often done in Java, but C++, C#, C, Prolog, Haskell, Perl, PHP, Ruby and Python also have packages to do so. Chapter 10 will show a few examples of handling XML files in different programming languages other than Java.

But before we explain how to program our compacting example in Java, it is important to understand different models of XML processing: *Document Object Model (DOM)*, *Simple API for XML (SAX)* and *Streaming API for XML (StAX)*.

The DOM programming model is similar to the one we have been using implicitly in previous chapters: by reading an XML file, a parser builds an internal tree structure file which it then traverses and modifies. Using a programming language without any restriction instead of a rule language such as XSLT, it is possible to destroy the tree structures with those manipulations so it is important to have a rigorous programming discipline.

Building the entire tree structure in memory before starting to process it can be prohibitive in the case of large XML files, so SAX, an alternative programming model, has been defined: as elements are parsed, user defined call-back procedures are invoked to do the processing. This requires much less memory because only a part of the document needs to be kept in memory at all time. However, the program is more limited in the kind of processing it can do efficiently or simply. This limitation is similar to the one observed between algorithms reading random-access files and those reading sequential-access files.

On top of the DOM and SAX approaches to parsing that have been available in the standard Java API since version 1.4, another *streaming* approach, introduced in Java 1.6, implements *pull parsing* that offers a relatively simple programming model and an efficient memory management. In this case, the program asks for the next XML token from the parser in order to select the appropriate action according to its type. So contrarily to the SAX approach in which the program is called back by the parser at each new token, in a pull-parser it is the program that controls the progression in the XML document. So processing of the file can be stopped as soon as it has been determined that the rest of the file is not relevant or it is possible to skip rapidly over irrelevant parts of the document for a certain application.

In this chapter, we will show how our compact pretty-print application shown in Section 5.3 can be implemented with each of the DOM, SAX and StAX programming models.

In Java, all XML document processing classes have been unified under the *Java API for XML Processing (JAXP)*, for controlling XML parsing, validation and transformation. It describes a unified interface independently of particular XML parsers and transformation engines.

8.1. Document Object Model (DOM)

The document object model is standardized by the W3C consortium, but its Java bindings can depend on the implementation. Java, since version 1.4, integrates XML processing packages that we use in our examples; no other special library is needed this way. The Java program given in Example 8.1 is a command line application that accepts an XML file as parameter and outputs the same compact text representation as the one used in Example 5.8 (page 80) that we obtained with the XSLT stylesheet of Example 5.9.

The first lines of Example 8.1 import the necessary packages to process the XML files. The `main` method (line 26-❶) creates a `DocumentBuilderFactory` object (line 34-❷) from which we will obtain a DOM parser after having configured the necessary options. By default, parsing only checks for well-formedness, so in order for the parsing to validate against a DTD a flag must be set (line 36-❸). To validate against an XML Schema another one must be set (line 38-❹). As explained in Section 3.1.1 and Section 3.5, the XML instances reference their corresponding DTD or XML Schema.

Creating a parser to build a new DOM document is done (line 41-❺) by using a factory method which returns a `DocumentBuilder` object to which an error handler (line 5-❶ of Example 8.2) is assigned to get a notification of possible error messages. If the file is valid (i.e. no `SAXParseException` is raised)¹, a `Document` object can be obtained and the `compact` method (line 83-❻) is called (line 46-❽) on the root element.

In order to simplify further processing, we define (line 66-❿) a method to remove from the DOM structure, starting from a given node, all text nodes containing only spaces and carriage returns and nodes other than element or text nodes, such as comments or processing instructions nodes. This method goes through each children removing nodes (line 76-⓫) that are not interesting for further processing; some care has to be taken to save the next sibling (line 71-⓬) before removing it from the DOM because the removal has the side-effect of setting the sibling nodes to null. For nodes that it does not remove, it calls itself recursively (line 78-⓭) to strip-space from the current child node. When it is called on the document element (line 45-⓮), it will go through all the document to remove empty text nodes and nodes that are not text or elements.

Within `compact` (line 83-⓯), the processing depends on the type of element obtained on line 85-⓰. If it is an element node (line 86-⓱), we first print the node name followed by an opening bracket. On line 91-⓲, attributes are printed with their names preceded by an @, followed by their value in square brackets. A new line is started if it is not the first attribute. The processing of the children elements, starting on line 97-⓳, is a simple traversal algorithm with a recursive call to `compact` (line 100-⓴), followed by the printing of a closing bracket. In the case of a text node (line 106-⓵), we print the content of the text node removing leading and trailing spaces, which could be carriage returns or newlines.

Example 8.1. [DOMCompact.java] Text compaction of the cellar book (Example 2.2) with Java using the DOM model

```
1 import org.w3c.dom.Attr;
  import org.w3c.dom.Document;
  import org.w3c.dom.NamedNodeMap;
  import org.w3c.dom.Node;
5
  import javax.xml.parsers.DocumentBuilder;
```

¹Even in the DOM model, `SAXExceptions` and `SAXParseException`s can be raised by the document builder.

```

import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.FactoryConfigurationError;
import javax.xml.parsers.ParserConfigurationException;
10 import org.xml.sax.InputSource;
import org.xml.sax.ErrorHandler;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
15 import java.io.IOException;

import javax.swing.JTree;
import javax.swing.JFrame;
20 import javax.swing.JScrollPane;
import javax.swing.tree.DefaultMutableTreeNode;
import javax.swing.tree.DefaultTreeModel;

public class DOMCompact{
25     public static void main(String argv[]) {
        // is there anything to do?
        if (argv.length != 1) {
            System.out.println("Usage: java DOMCompact file");
30             System.exit(1);
        }
        // parse file
        try {
            DocumentBuilderFactory factory =
35             DocumentBuilderFactory.newInstance();
            factory.setValidating(true);
            factory.setNamespaceAware(true);
            factory.setAttribute(
40             "http://java.sun.com/xml/jaxp/properties/schemaLanguage",
            "http://www.w3.org/2001/XMLSchema");
            DocumentBuilder builder = factory.newDocumentBuilder();
            builder.setErrorHandler(new CompactErrorHandler());
            Document doc = builder.parse(argv[0]);
            Node docElem=doc.getDocumentElement();
45             stripSpace(docElem);
            compact(docElem, "");
            System.out.println();
            new TreeViewer(
50             new JTree(new DefaultTreeModel(
                TreeViewer.jTreeBuild(docElem)))).setVisible(true);
        } catch (SAXParseException e) {
            System.out.println(argv[0]+"is not well-formed");
            System.out.println(e.getMessage()+"at line "+e.getLineNumber()+
                ", column "+e.getColumnNumber());
55         } catch (SAXException e){
            System.out.println(e.getMessage());
        } catch (ParserConfigurationException e){
            System.out.println("Parser configuration error");
        } catch (IOException e) {
60             System.out.println("IO Error on "+argv[0]);

```

```
    }
}

// remove empty text nodes (ie nothing else than spaces and carriage return)
65 // and nodes that are not text or element ones
    private static void stripSpace(Node node){ 12
        Node child = node.getFirstChild();
        while(child!=null){
            // save the sibling of the node that will
70 // perhaps be removed and set to null
            Node c = child.getNextSibling(); 13
            if((child.getNodeType() != Node.TEXT_NODE &&
                child.getNodeValue().trim().length() == 0) ||
                ((child.getNodeType() != Node.TEXT_NODE) &&
75 (child.getNodeType() != Node.ELEMENT_NODE)))
                node.removeChild(child); 14
            else // process children recursively 15
                stripSpace(child);
                child=c;
80     }
    }

public static void compact(Node node, String indent) { 16
    if (node == null) return;
85     switch (node.getNodeType()) { 17
        case Node.ELEMENT_NODE: { 18
            System.out.print(node.getNodeName()+ '[');
            indent += blanks(node.getNodeName().length()+1);
            NamedNodeMap attrs = node.getAttributes();
90             boolean first=true;
            for (int i = 0; i < attrs.getLength(); i++) { 19
                if(!first) System.out.print('\n'+indent);
                System.out.print('@'+attrs.item(i).getNodeName()+ '['
95                 +attrs.item(i).getNodeValue()+']');
                first=false;
            }
            for(Node child = node.getFirstChild(); 20
                child != null; child = child.getNextSibling()){
                if(!first) System.out.print('\n'+indent);
100                compact(child, indent); 21
                first=false;
            }
            System.out.print(']');
            break;
105     }
        case Node.TEXT_NODE: { 22
            System.out.print(node.getNodeValue().trim());
            break;
        }
110     }
}

// production of string of spaces with a lazy StringBuffer
private static StringBuffer blanks = new StringBuffer();
```

```

115     private static String blanks(int n){                                23
        for(int i=blanks.length();i<n;i++)
            blanks.append(' ');
        return blanks.substring(0,n);
    }
120 }

```

- ❶ Beginning of the Java program that checks if there is a parameter to be used as a file name to compact.
- ❷ Beginning of the parsing process creating a document builder that will be used for validation and for creating the DOM structure of the document.
- ❸ Indicates that document validation will be performed, on top of the default well-formedness check.
- ❹ Validation should be done with a Schema, not a DTD.
- ❺ Creates a class that will be used for creating the DOM structure.
- ❻ Parses the document using the previously created objects.
- ❼ Saves a reference to the root document element.
- ❽ Remove all white space nodes and non text or element nodes from the whole DOM structure.
- ❾ Creates a compact form by traversing the DOM structure from the root. `compact` is defined on line 83-❿
- ❿ Creates an interactive window representing the DOM structure.
- ⓫ Deals with exceptions that can occur during the parsing phase, such as non well-formedness, validation error or file-access exceptions.
- ⓬ Empty text children nodes are removed so that only meaningful content nodes are compacted.
- ⓭ Saves a reference to the next sibling.
- ⓮ Removes the current child from the DOM structure.
- ⓯ Strips space recursively on this child node.
- ⓰ Prints a compact representation of the current node. `indent` is a string that is added to the start of each line of output in this node.
- ⓱ Determines the type of the current node in order to choose the appropriate processing step.
- ⓲ An element node is first printed followed by an opening bracket and the indentation is updated so that recursive calls will print their content more indented than the name of the current node.
- ⓳ Attributes are printed on separate (indented) lines; an attribute name is prefixed with an @ and the value is enclosed in square brackets.
- ⓴ The children are processed with the current indentation. Finally, a closing bracket is added.
- ⓵ Compact this child recursively.
- ⓶ Normalizes and prints a text node.
- ⓷ Returns a string of a given number of spaces as a substring of static `StringBuffer` that is expanded as necessary.

Example 8.2. [CompactErrorHandler.java] Error handler of the DOM parsing of Example 8.1

```

1 import org.xml.sax.ErrorHandler;
import org.xml.sax.SAXException;                                ❶
import org.xml.sax.SAXParseException;

5 public class CompactErrorHandler implements ErrorHandler{      ❷
    private void message(String mess,SAXParseException e)      ❸
        throws SAXException{
        System.out.println("\n"+mess+

```

```
10         "\n Line:" + e.getLineNumber() +
           "\n URI:" + e.getSystemId() +
           "\n Message:" + e.getMessage());
    }

    public void fatalError(SAXParseException e) throws SAXException{ ❹
15         message("Fatal error", e);
    }

    public void error(SAXParseException e) throws SAXException{      ❺
20         message("Error", e);
    }

    public void warning(SAXParseException e) throws SAXException{    ❻
25         message("Warning", e);
    }
}
```

- ❶ SAXException and SAXParseException are used even though DOM parsing is used.
- ❷ Implementation of XML parsing error handling.
- ❸ Prints out an error message with the current error position. This method is used for all three kinds of errors.
- ❹ An irrecoverable error.
- ❺ An XML validation error.
- ❻ A simple warning not requiring processing to stop.

8.2. Simple API for XML (SAX)

In the SAX model of processing, the system calls event-handler methods as it parses the file. As the whole document does not have to be kept in main memory, this is quite memory-efficient but then global variables have to be used for communicating between handler methods. In our case, the only global information needed are the current indentation and the fact that the current line be ended.

The current line should be ended at the start of a new element only when the last thing printed was a closing bracket. Many closing brackets can be put on the same line though. So we keep a shared boolean variable for this state and a shared integer storing the current number of blank spaces used for indentation.

Creating a SAX parser is done via a *factory* in much the same way as we have shown in the previous section for a DOM parser. Example 8.3 describes the main procedure which creates a factory (line 29-❸) and sets flags for validation. The parser is obtained on line 32-❹ and a property is set to indicate that we want validation to be done with an XML Schema. Parsing is then started on line 36-❺ by passing a reference to an Handler object which receives call-backs during the parsing process.

Example 8.3. [SAXCompact.java] Text compaction of the cellar book (Example 2.2) with Java using the SAX model

```

1 import org.xml.sax.SAXException;
  import org.xml.sax.SAXParseException;
  import org.xml.sax.helpers.XMLReaderFactory;

5 import org.xml.sax.helpers.DefaultHandler;
  import javax.xml.parsers.SAXParserFactory;
  import javax.xml.parsers.ParserConfigurationException;
  import javax.xml.parsers.SAXParser;

10 import java.io.IOException;

  import javax.swing.JTree;
  import javax.swing.JFrame;
  import javax.swing.JScrollPane;
15 public class SAXCompact {

    private static JTree jtree = new JTree();                                ❶

20 public static void main(String argv[]) {                                  ❷
    if (argv.length != 1) {
        System.out.println("Usage: java SAXCompact file");
        return;
    }
25 //XMLParser creation
    SAXParserFactory factory;
    SAXParser saxParser;
    try {
30         factory = SAXParserFactory.newInstance();                          ❸
        factory.setNamespaceAware(true);
        factory.setValidating(true);
        saxParser = factory.newSAXParser();                                  ❹
        saxParser.setProperty(
35             "http://java.sun.com/xml/jaxp/properties/schemaLanguage",
            "http://www.w3.org/2001/XMLSchema");
        // parse file and print compact form                                ❺
        saxParser.parse(argv[0], new CompactHandler());
        System.out.println();
        // parse file and build a tree form
40         saxParser.parse(argv[0], new JTreeHandler(jtree));                ❻
        // display the built tree
        new TreeViewer(jtree).show();                                       ❼
    } catch (ParserConfigurationException e) {                               ❽
        System.out.println("Bad parser configuration");
45     } catch (SAXParseException e) {
        System.out.println(argv[0]+" is not well-formed");
        System.out.println(e.getMessage()+
            " at line "+e.getLineNumber()+
            ", column "+e.getColumnNumber());
50     } catch (SAXException e){

```

```
        System.out.println(e.getMessage());
    } catch (IOException e) {
        System.out.println("IO Error on "+argv[0]);
    }
55     } // main(String[])
    } // class SAXCompact
```

- ❶ Initialises an empty `JTree` for interactive output.
- ❷ Main method that first checks if a file name has been given as an argument to the program.
- ❸ Creates a document factory.
- ❹ Creates a new SAX parser.
- ❺ Parses the file given as argument while sending the parse events to a handler that will output the compact form.
- ❻ Parses the file given as argument while sending the parse events to a handler that will add nodes to a `JTree` for interactive viewing.
- ❼ Displays the interactive tree viewer.
- ❽ Handles exceptions that can occur in both parsing and input file processing.

Example 8.4 shows the structure of a SAX event handler (a subclass of the `DefaultHandler` class), which defines empty handlers for all type of events including errors. In our case, only `startElement` (line 20-❶), `endElement` (line 38-❷) and `characters` (line 45-❸) are called when encountering text nodes.

When an error is encountered during the parsing process, one of the methods defined on lines starting on line 66-❹ is called. When this happens, we call the `message` method (line 58-❶) which prints some useful information about the error.

Example 8.4. [`CompactHandler.java`] SAX handler for text compacting an XML file such as Example 2.2

```
1 import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;
import org.xml.sax.Attributes;
import org.xml.sax.SAXParseException;
5
public class CompactHandler extends DefaultHandler{

    // production of string of spaces with a lazy StringBuffer
private static StringBuffer blanks = new StringBuffer();
10 private String blanks(int n){
    for(int i=blanks.length();i<n;i++)
        blanks.append(' ');
    return blanks.substring(0,n);
}
15
private boolean closed = false; // closed mode?
protected int indent; // current indentation value
```



```

20     public void startElement(String uri, String localName,           ❶
        String raw, Attributes attrs)
        throws SAXException {
        if(closed){
            System.out.print('\n'+blanks(indent));
25         closed = false;
        }
        indent=indent+1+localName.length();
        System.out.print(localName+'[');
        // deal with attributes
30     for (int i = 0; i < attrs.getLength(); i++) {
            if(i>0)System.out.print('\n'+blanks(indent));
            System.out.print('@'+attrs.getLocalName(i)+'['
                +attrs.getValue(i)+'']');
            closed=true;
35     }
    }

    public void endElement(String uri, String localName, String raw) ❷
        throws SAXException {
40     System.out.print(']');
        closed = true;
        indent=indent-1-localName.length();
    }

45     public void characters(char[] ch, int start, int length)       ❸
        throws SAXException {
        if(closed){
            System.out.print('\n'+blanks(indent));
            closed = false;
50     }
        String s = new String(ch,start,length).trim();
        System.out.print(s);
        if(s.length()>0)
            closed=true;
55     }

    // error handling...
    private void message(String mess,SAXParseException e)           ❹
        throws SAXException{
60     System.out.println("\n"+mess+
            "\n Line:"+e.getLineNumber()+
            "\n URI:" +e.getSystemId()+
            "\n Message:"+e.getMessage());
    }

65     public void fatalError(SAXParseException e) throws SAXException{ ❺
        message("Fatal error",e);
    }

70     public void error(SAXParseException e) throws SAXException{
        message("Error",e);
    }

```

```
75     public void warning(SAXParseException e) throws SAXException{  
        message("Warning", e);  
    }  
}
```

- ❶ Checks whether the current line should be terminated. Then the name of the current element and an opening bracket are printed. The shared indentation value is updated and the attributes are output if there are any.
- ❷ A closing bracket is printed and the indentation value is decreased by the length of the name. Because the file has been validated during the parsing process, it is sure that the `localName` variable is the same as the one used to increase the indentation in the corresponding start-tag method.
- ❸ The characters are printed after having removed leading and trailing whitespace. Note that the `characters` method is not called with a `String` but with an array of characters as well as a start position and the number of characters to use from the character array. This can sometimes avoid the allocation of a new `String` for each element.
- ❹ Method used to output error messages.
- ❺ Error handling for fatal, validation and recoverable errors.

8.3. Stream API for XML (StAX)

StAX, like SAX, is stream-oriented but in StAX, the programmer is *in control* using a simple API based on the notion of a cursor that walks the XML document from beginning to end. The cursor can only move forward and points to a single point in the XML file². Only a small part of the whole XML document needs to reside in memory at any single time, so this approach is quite memory-efficient, as it is for SAX. But as the programmer controls which methods are called when a given token is at the cursor, it is possible to add contextual information to each call for dealing with a token without the need of global variables as in SAX.

Creating a StAX parser is done via a *factory* in much the same way as we have shown in the previous sections for DOM or SAX parsers. Example 8.5 describes the main procedure which creates a factory (line 25-❷). Then we configure the parser by setting some properties: here we want entities to be interpreted by the parser instead of being passed directly to our program and we would like to receive consecutive character nodes as a single concatenation of all these nodes. Parsing is done using an `XMLStreamReader` created (line 28-❸) from the `XMLInputFactory`. This reader will be an iterator-like object that gets a new token at each call to `next()` which returns its type as an integer. This type can then be checked for equality with predefined constants in the `XMLEvent` class but, in our program, we prefer to use parser utility functions such as `isStartElement()` or `isCharacters()` that check the type of the current token. In Example 8.5, we only deal with the start or end of an element or with its character content.

Similarly to the DOM approach in Example 8.1, the compacting process follows recursively the tree structure. After skipping what comes before the first element, we call the `compact` method (line 31-❹) with the current state of the XML stream which will process the current element and its children before returning.

Within `compact` (line 47-❶), the type of current token is checked and if it is the starting tag of an element (line 49-❷) then the name of the element is printed and the indentation is updated. Then attributes and values are output properly indented (line 53-❸). Each non empty children is then printed by a recursive call to

²There is also a StAX iterator based API that represents the XML document as a set of discrete events that are pulled by the application in the order in which they show up in the document, but we will not consider this approach here.

compact (line 65-⑩). A closing bracket is output when the end element tag is encountered which terminates the loop. The character content (line 68-⑪) is simply printed as is.

Example 8.5. [StAXCompact.java] Text compaction of the cellar book (Example 2.2) with Java using the StAX model

```

1 import java.io.FileInputStream;
  import java.io.IOException;

  import javax.swing.JTree;
5 import javax.swing.tree.DefaultTreeModel;
  import javax.xml.stream.XMLInputFactory;
  import javax.xml.stream.XMLStreamException;
  import javax.xml.stream.XMLStreamReader;

10 public class StAXCompact {
    // production of string of spaces with a lazy StringBuffer
    private static StringBuffer blanks = new StringBuffer();
    private static String blanks(int n){
15         for(int i=blanks.length();i<n;i++)
            blanks.append(' ');
        return blanks.substring(0,n);
    }

    public static void main(String[] argv) {                                ①
20         if (argv.length != 1) {
            System.out.println("Usage: java StAXCompact file");
            return;
        }
        try {
25             XMLInputFactory xmlif = XMLInputFactory.newInstance();        ②
            xmlif.setProperty(XMLInputFactory.IS_REPLACING_ENTITY_REFERENCES, true);
            xmlif.setProperty(XMLInputFactory.IS_COALESCING, true);
            XMLStreamReader xmlsr =                                          ③
30                 xmlif.createXMLStreamReader(argv[0], new FileInputStream(argv[0]));
            while(!xmlsr.isStartElement())xmlsr.next();                    ④
            compact(xmlsr,"");                                             ⑤
            System.out.println(); // end last line
            xmlsr.close();                                                 ⑥
            // restart to create the JTree
35             xmlsr=xmlif.createXMLStreamReader(argv[0], new FileInputStream(argv[0]));
            while(!xmlsr.isStartElement())xmlsr.next();
            new TreeViewer(new JTree(
                new DefaultTreeModel(TreeViewer.jTreeBuild(xmlsr))        ⑦
            )).setVisible(true);
40         } catch (XMLStreamException ex) {                                ⑧
            System.out.println(ex.getMessage());
        } catch (IOException e) {
            System.out.println("IO Error on "+argv[0]);
        }
45     }

    private static void compact(XMLStreamReader xmlsr,String indent) ⑨

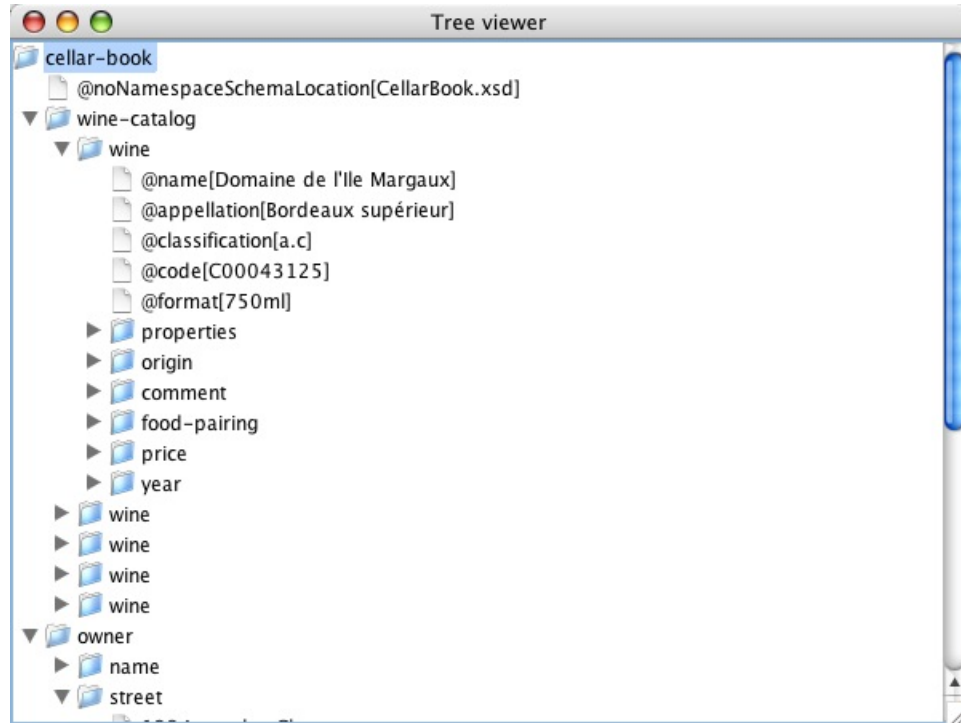
```

```
        throws XMLStreamException{
    if(xmlsr.isStartElement()){
50         String localName = xmlsr.getLocalName();
           System.out.print(localName+'[');
           indent += blanks(localName.length()+1);
           int count = xmlsr.getAttributeCount(); // attributes
           for (int i = 0; i < count; i++) {
55             if(i>0)System.out.print('\n'+indent);
               System.out.print("@"+xmlsr.getAttributeLocalName(i)+
                   '['+xmlsr.getAttributeValue(i)+'']');
           }
           boolean first=count==0;
60           while (true){
               do {xmlsr.next();} while(xmlsr.isWhiteSpace());
               if (xmlsr.isEndElement()) break;
               if (first) first=false;
               else System.out.print('\n'+indent);
65               compact(xmlsr,indent);
           }
           System.out.print(']');
           } else if (xmlsr.isCharacters()){
70             System.out.print(xmlsr.getText());
           }
       }
    }
```

- ❶ Main method that first checks if a file name has been given as an argument to the program.
- ❷ Creates an input factory.
- ❸ Creates a new stream parser.
- ❹ Ignores the tokens that come before the first element (e.g. processing instructions or DTD).
- ❺ Calls the compacting process with the current token and then prints a newline to flush the content of the last line.
- ❻ Releases the data structure of the last streaming operation before opening the file for the second time, to build the JTree.
- ❼ Handles exceptions that can occur in both parsing and input file processing.
- ❽ Method to compact from the current token.
- ❾ If it is a start element tag, outputs the name of the element followed by an opening bracket and update the current indentation.
- ❿ Outputs each attribute name and value all indented except the first one.
- ⓫ Loops on children nodes that are not whitespace and compacts each of them with the correct indentation.
- ⓬ Recursive call to the compacting process.
- ⓭ Prints the character content.

8.4. Showing an Interactive Tree View

The Java API already provides a graphical view of trees with the `JTree` class. It displays the nodes in a window and they can be expanded and collapsed by clicking on their handles. This kind of display can also be obtained by XSL-FO (see section 6.9 of [7]) but few systems currently implement the full specification

Figure 8.1. JTree display of Example 2.2

which would allow this to happen. Internet Explorer (top right of Figure 1.2) and Firefox use a similar scheme when displaying XML files but it is based on the `visible` property of Cascading Style Sheets (CSS). This form of interaction is used on most operating systems to display the contents of directories. For example, the XML file of Example 2.2 (page 10) can be displayed in a window like the one shown in Figure 8.1. Nodes are shown there with directory icons and can be expanded or collapsed by clicking on the triangle to the left of the icon. A node showing a collapsed subtree has a triangle that points downward when it is expanded. A different display can be obtained by changing the *look and feel* in the Java API but the principle stays the same.

8.4.1. Building a JTree with DOM

Creating such a view from a DOM structure is only a matter of traversing the structure to create nodes that will be part of the JTree display. Its nodes are instances of the predefined `DefaultMutableTreeNode` class. To obtain the display in Figure 8.1, Example 8.1 (line 48-⑩) creates a new `TreeViewer` instance that displays a JTree using a tree model built from the DOM tree constructed using the `jTreeBuild` method. Example 8.6 defines the class `TreeViewer` (line 13-①) that creates a window to display the JTree instance. It is assigned an initial constant position and size and made scrollable; the application should terminate when the window is closed.

`jTreeBuild` (line 21-②) is a static method so it is called with the instruction `TreeViewer.jTreeBuild(.)`, (line 21-② and line 48-⑩ of Example 8.2). It uses the same algorithm as `compact` (line 83-⑥ of Example 8.1) by recursively processing *element* or *text* DOM nodes. In the case of an element node (line 25-③), it creates a new `DefaultMutableTreeNode` for the element and adds attributes as its first child; it then processes each child by recursively building its subtree (line 37-④) which

is added as a child of the current node. A non-empty text node is simply a `DefaultMutableTreeNode` having the text as label.

Example 8.6. [TreeViewer.java]: JTree building with DOM and StAX Processing of an XML file

```
1 import javax.swing.JFrame;
  import javax.swing.JScrollPane;
  import javax.swing.JTree;
  import javax.swing.tree.DefaultMutableTreeNode;
5
  import javax.xml.stream.XMLStreamException;
  import javax.xml.stream.XMLStreamReader;
  import javax.xml.stream.events.XMLEvent;

10 import org.w3c.dom.NamedNodeMap;
   import org.w3c.dom.Node;

   public class TreeViewer extends JFrame{                                ❶
       TreeViewer(JTree jtree) {
15         super("Tree viewer");
           setBounds(100,100,600,450);
           getContentPane().add(new JScrollPane(jtree));
           setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
       }

20   public static DefaultMutableTreeNode jTreeBuild(Node node) {        ❷
       if (node == null)
           return null;
       switch (node.getNodeType()) {
25         case Node.ELEMENT_NODE: {                                     ❸
           DefaultMutableTreeNode treeNode =
               new DefaultMutableTreeNode(node.getNodeName());
           NamedNodeMap attrs = node.getAttributes();
           for (int i = 0; i < attrs.getLength(); i++)
30             treeNode.add(
                 new DefaultMutableTreeNode(
                     '@'+attrs.item(i).getNodeName()+
                     '['+attrs.item(i).getNodeValue()+']');
           // process children
35           Node child = node.getFirstChild();
           while(child != null){
               DefaultMutableTreeNode childTree = jTreeBuild(child); ❹
               if(childTree!=null)
                   treeNode.add(childTree);                             ❺
40             child = child.getNextSibling();
           }
           return treeNode;
       }
       case Node.TEXT_NODE: {
45         String text = node.getNodeValue().trim();
           return text.length()==0 ? null                               ❻
               : new DefaultMutableTreeNode(text);
```

```

    }
    default : return null; // ignore other types of nodes
50    }
}

public static DefaultMutableTreeNode jTreeBuild      ⑦
    (XMLStreamReader xmlsr) throws XMLStreamException{
55    if(xmlsr.isStartElement()){                      ⑧
        DefaultMutableTreeNode treeNode
            = new DefaultMutableTreeNode(xmlsr.getLocalName());
        int count = xmlsr.getAttributeCount();// attributes    ⑨
        for (int i = 0; i < count; i++) {
60            treeNode.add(
                new DefaultMutableTreeNode(
                    "@"+xmlsr.getAttributeLocalName(i)+
                    '['+xmlsr.getAttributeValue(i)+'']');
        }
65        while (true){                               ⑩
            do {xmlsr.next();} while(xmlsr.isWhiteSpace());
            if(xmlsr.getEventType()==XMLEvent.END_ELEMENT)break;
            DefaultMutableTreeNode childTree = jTreeBuild(xmlsr);
            if(childTree!=null) treeNode.add(childTree);
70        }
        return treeNode;
    } else if (xmlsr.isCharacters()){                  ⑪
        return new DefaultMutableTreeNode(xmlsr.getText());
    } else
75    return null;
}
}

```

- ① Inherits from JFrame so that the tree is displayed in a new window. The content of the constructor describes the position and dimensions of the window, inserts it in a scrolling pane and makes the program end when the window is closed.
- ② Recursive method for creating a JTreeNode from a document node.
- ③ When the node is an element node, creates a new JTree node to which are added the attributes as simple JTree nodes and the children nodes which are built recursively.
- ④ Recursive building of the tree corresponding to a child.
- ⑤ Adds the new children JTree node as a child of the current JTree node.
- ⑥ In the case of a non-empty text node, creates a simple JTree node.
- ⑦ Recursive method for creating a JTreeNode from tokens returned by an XMLStreamReader.
- ⑧ Creates a new tree node with the name of the element.
- ⑨ Adds a node for each attribute name and value.
- ⑩ Loops on children nodes that are not whitespace and adds a new child for each new tree built.
- ⑪ Creates a node with the character content.

8.4.2. Building a JTree with SAX

A JTree using SAX processing is created with a special-purpose handler whose methods will be called when XML elements are encountered during the parsing process. This is similar to the process described in Section 8.2 where we only needed to keep the current indentation value.

In Example 8.7, we build a tree using `DefaultMutableTreeNode` instances and keep track of the current node being built (line 13-❶). When an XML element start-tag is encountered, in `startElement` (line 19-❸), a new tree node is created and made the current node (line 25-❹). Its attributes are then added as children (line 29-❺). A text node is simply added as a child of the current node (line 41-❻). When an end-tag is encountered (line 34-❼), we set the current node to the parent of the current node. The constructor (line 15-❷) only keeps a copy of the `JTree` that will be displayed. Its model is initialized on the first call of `startElement` (line 22-❽).

To create a `JTree`, we add (line 40-❾ of Example 8.3) a new `JTreeHandler` given as argument to the SAX parser. Display of the tree is achieved by creating an instance (line 42-❿) of the `TreeViewer` class (line 13-❶ of Example 8.6). The `JTree` instance is built by calling the SAX parser and giving it a `JTreeHandler` (line 40-❾).

SAX allows the *pipelining* of handlers that work in succession during a single parse of the file, a process called *filtering*, but here we simplify by parsing the file twice: once for the textual input (line 36-❶ of Example 8.3) and once more for the tree display (line 40-❾).

Example 8.7. [`JTreeHandler.java`]: `JTree` building with SAX Processing of an XML file

```
1 import org.xml.sax.SAXException;
  import org.xml.sax.Attributes;
  import org.xml.sax.helpers.DefaultHandler;

5 import javax.swing.JTree;
  import javax.swing.tree.DefaultMutableTreeNode;
  import javax.swing.tree.DefaultTreeModel;

  public class JTreeHandler extends DefaultHandler{
10
    private DefaultTreeModel treeModel;
    private DefaultMutableTreeNode node = null;
    private JTree jtree;                                ❶

15    JTreeHandler(JTree jtree){                          ❷
        this.jtree=jtree;
    }

    public void startElement(String uri, String localName,
20        String raw, Attributes attrs) throws SAXException {  ❸
        super.startElement(uri,localName,raw,attrs);          ❹
        if(node==null) //initialise node and model            ❺
            jtree.setModel(
                new DefaultTreeModel(
25                node=new DefaultMutableTreeNode(localName));  ❻
            else // add child and set node to the child
                node.add(node=new DefaultMutableTreeNode(localName));
            // add attributes as children
            for (int i = 0; i < attrs.getLength(); i++)          ❼
30                node.add(new DefaultMutableTreeNode(
                    '@'+attrs.getLocalName(i)+"["+attrs.getValue(i)+'']');
        }
    }
```



```

    public void endElement(String uri, String localName, String raw) ❸
35     throws SAXException {
        super.endElement(uri, localName, raw); ❹
        node = (DefaultMutableTreeNode) node.getParent();
    }

40     public void characters(char[] ch, int start, int length) ❺
        throws SAXException { ❻
            super.characters(ch, start, length);
            String text = new String(ch, start, length).trim();
            if(text.length()>0) node.add(new DefaultMutableTreeNode(text));
45     }
    }

```

- ❶ The JTree node currently being built.
- ❷ Initializes the JTree structure to which the JTree nodes will be added.
- ❸ Deals with a start element.
- ❹ Calls the start method of the superclass.
- ❺ If the node is not initialized, creates a new tree model and uses it for the root.
- ❻ Creates a new node and adds it to the current JTree node. Note that the new current node is updated to this new node.
- ❼ Adds attributes as nodes of the current node.
- ❽ Processes the end tag.
- ❾ Calls the endElement method of the superclass and the current node is now the parent of the current node.
- ❿ Processes a text node by creating a new JTree node with text as its content if it is not empty.

8.4.3. Building a JTree with StAX

A JTree using StAX processing is created with the same recursive process described for the DOM approach. Looking at Example 8.6, we immediately see the strong similarities between the two approaches. It is only a matter of traversing the structure to create nodes that will be part of the JTree display. Its nodes are instances of the predefined DefaultMutableTreeNode class. To obtain the display in Figure 8.1, Example 8.5 (line 38-❷) creates a new TreeViewer instance that displays a JTree by parsing again the file, this is why we create a new instance of a XMLStreamReader that is given as parameter to the second TreeViewer.JTreeBuild static method in Example 8.6

It uses the same algorithm as compact (line 83-❹ of Example 8.1) by recursively processing *element* or *text* DOM nodes. In the case of an element node (line 55-❸), it creates a new DefaultMutableTreeNode for the element and adds attributes as its first children (line 65-❿); it then processes each child by recursively building its subtree (line 65-❿) which is added as a child of the current node. In the case of character content (line 72-❶), a single node with this character content is returned.

8.5. Additional Information on Programming Models

The Java API since version 1.4 includes classes for the DOM and SAX approaches and for StAX since 1.6. XML processing is now described into some introductory Java books such as *Big Java* [25].

XML in a Nutshell [37] provides a short but thorough presentation of these programming models.

Sun publishes excellent online tutorials: [5] for the DOM and SAX processing models and chapter 3 of [72] for the StAX approach. *Java & XML* [36] is a very good information source for the creation and manipulation of XML documents in Java.

In this chapter, we have used Java to access the content of an XML file, but there are XML parsers for almost every computer languages such as C [62], C++ [64], C# [63], Perl [42] [40], PHP [77], Python [40], Haskell [50], Prolog [53], Ruby [80], even COBOL [21].

In our case, we were dealing similarly with all document nodes without ever looking at the names of the elements. As it often happens that the information needed in a document is deep down in the tree, one must then access specific nodes using a sequence of `getFirstChild()` and `getNextSibling()` calls until we get to the right element. This process must often be repeated at each level while it was more or less implicit using templates and XPath. So we often define special purpose functions and methods to traverse a specific document tree, but there are other ways. For example, XMLSpy can generate automatically access methods, in either C++ or Java, from the the XML Schema that validates a document. This is very useful and much less error prone than relying on a hand-coded traversal of the tree. When the schema is changed, then methods can be regenerated and processing of a valid XML will always follow the structure of the XML Schema. There are also tools to convert database schemas into XML Schema. So we see that XML processing can be easily integrated with other systems and languages.

Chapter 9. Document Creation by Programming in Java

In the previous section, we have shown how to parse an existing XML document by programming. It is also important to see how an XML document can be created by programming. We will therefore show the *inverse* of the programs shown previously by writing a program that parses the *compact* form we produced and expand it into the corresponding XML document. In principle, after compaction and expansion, we should recover the original XML document with which we started but since we have not faithfully transformed whitespace, the files are not strictly identical. Although one can write a program to create a file with XML tags and their content using `print` methods, we will show (Section 9.1) that it is both simpler and more systematic to create an XML document in memory using the DOM model, to modify it and then to print it using a serializer. In Section 9.2, we will describe how to create an XML document by parsing a text file and send SAX events to a transformer and in Section 9.3 we will show how to do the same with the StAX streaming approach.

9.1. Creating a DOM Document

The DOM API provides an exhaustive collection of methods to create and modify a document. The most frequently used are:

<code>builder.newDocument()</code>	creates an empty document to which elements can be added
<code>doc.createElement(String s)</code>	creates a new element named <code>s</code> in document <code>doc</code>
<code>parent.appendChild(Element e)</code>	adds element <code>e</code> as the last child of element <code>parent</code> ; if <code>e</code> was already the child of another node, it is removed before being assigned to its new parent
<code>e.setAttribute(String name, String value)</code>	adds the attribute name with the corresponding value to the element <code>e</code> ; if the attribute already exists, its value is replaced
<code>doc.createTextNode(String s)</code>	creates a new text node with content <code>s</code> in document <code>doc</code>

In order to simplify parsing, we create a customized `StreamTokenizer` which returns a single token for all characters between separators used in the compact form (i.e. opening and closing brackets, at-sign and newline). The separators are also returned as a single token. The implementation of this tokenizer is given in Example 9.1.

In Example 9.2, the `main` method (line 24-❶) first creates a `Document` instance (line 29-❷) which will hold the XML tree. It then creates a specialized tokenizer (line 30-❸) from the file name given as argument to the program. It goes on to find the name of the root element (line 34-❹) and creates the root node. It then calls the `expand` method (line 39-❺) which add the whole content of the element a child of the document. To output the DOM structure, we create an *identity* transformation (line 41-❻) and use the document as source and `System.out` as output (line 46-❼). We also set an *output property* so that the output is nicely indented (line 43-❼).

Expansion (line 52-❶) is a recursive process that adds element and text nodes to a parent node received as a parameter; it processes each attribute (line 55-❷) by getting the name and value of the attribute and adding it to the current element line 58-❸; the content of the element (line 62-❹) is processed by looping until a closing bracket is encountered. When the next node is followed by an open bracket, a new child is created and expanded recursively (line 66-❺); otherwise, a text node is created (line 68-❻).

Example 9.1. [CompactTokenizer.java]: Specialized stream tokenizer that ignores blank tokens

```
1 import java.io.Reader;
import java.io.IOException;
import java.io.StreamTokenizer;

5 public class CompactTokenizer {
    private StreamTokenizer st;

    CompactTokenizer(Reader r){ ❶
        st = new StreamTokenizer(r);
10     st.resetSyntax(); // remove parsing of numbers... ❷
        st.wordChars('\u0000', '\u00FF'); // everything is part of a word
        // except the following...

        st.ordinaryChar('\n');
        st.ordinaryChar('[');
15     st.ordinaryChar(']');
        st.ordinaryChar('@');
    }

    public String nextToken() throws IOException{ ❸
20     st.nextToken();
        while(st.ttype=='\n' ||
            (st.ttype==StreamTokenizer.TT_WORD &&
             st.sval.trim().length()==0))
            st.nextToken();
25     return getToken();
    }

    public String getToken(){ ❹
30     return (st.ttype == StreamTokenizer.TT_WORD) ? st.sval : (""+(char)st.ttype);
    }

    public String skip(String sym) throws IOException { ❺
        if(getToken().equals(sym))
            return nextToken();
35     else
        throw new IllegalArgumentException("skip: "+sym+" expected but"+
            sym +" found ");
    }
}
40
```

- ❶ Constructor that receives a Reader and creates a customized StreamTokenizer.
- ❷ Because we do not want to have numbers and Java comments to be dealt with, we *reset the syntax* to indicate that all characters can be part of a word except for special separators used in the compact form.
- ❸ Calls the Java tokenizer and skips newlines and empty text nodes. It returns the current token as a String.
- ❹ Function for accessing the current token as a String.

- ⑥ Function for checking that the current token corresponds to the one given in parameter. Raises an exception if this is not the case. This is useful for checking the input that will not appear in the output corresponds to what is expected.

Example 9.2. [DOMExpand.java]: Compact form parsing to create a DOM XML document

A sample input for this program is Example 5.8 to give back Example 2.2.

```

1  import  org.w3c.dom.Element;
   import  org.w3c.dom.Document;
   import  javax.xml.parsers.DocumentBuilder;
   import  javax.xml.parsers.DocumentBuilderFactory;
5
   import  javax.xml.transform.OutputKeys;
   import  javax.xml.transform.Transformer;
   import  javax.xml.transform.TransformerFactory;
   import  javax.xml.transform.TransformerException;
10  import  javax.xml.transform.TransformerConfigurationException;
   import  javax.xml.transform.dom.DOMSource;
   import  javax.xml.transform.stream.StreamResult;

15  import  java.io.IOException;
   import  java.io.BufferedReader;
   import  java.io.FileInputStream;
   import  java.io.InputStreamReader;
   import  java.io.StreamTokenizer;
20
   public class DOMExpand {
       static CompactTokenizer st;

       public static void main( String[] argv ) {
25         try {
           DocumentBuilderFactory factory =
               DocumentBuilderFactory.newInstance();
           DocumentBuilder builder = factory.newDocumentBuilder();
           Document doc = builder.newDocument();
30         st = new CompactTokenizer(
               new BufferedReader(
                   new InputStreamReader(
                       new FileInputStream(argv[0]))));
           String rootName = "dummyElement";
35         // ignore everything preceding the word before the first "["
           while(!st.nextToken().equals("[")){
               rootName=st.getToken();
           }
           expand((Element)doc.appendChild(doc.createElement(rootName).trim()));
40         // output with an "identity" Transformer
           TransformerFactory tFactory = TransformerFactory.newInstance();
           Transformer transformer = tFactory.newTransformer();
           transformer.setOutputProperty(OutputKeys.INDENT, "yes");
           DOMSource source = new DOMSource(doc);
           transformer.transform(source, new StreamResult(doc));
       }
   }

```

```
45         StreamResult result = new StreamResult(System.out);
           transformer.transform(source,result);           ❸
           } catch ( Exception ex ) {
               ex.printStackTrace();
           }
50     }

           static void expand(Element elem) throws IOException{           ❹
               Document doc = elem.getOwnerDocument();           ❷
               st.skip("[");
55         while(st.getToken().equals("@")){// process attributes           ❶
               String attName = st.nextToken();
               st.nextToken();
               elem.setAttribute(attName,st.skip("["));           ❺
               st.nextToken();
60         st.skip("]");
           }
           while(!st.getToken().equals("]")){ // process element           ❸
               String s = st.getToken().trim();
               st.nextToken();
65         if(st.getToken().equals("[")
               expand((Element)elem.appendChild(doc.createElement(s))           ❻);
               else{
                   elem.appendChild(doc.createTextNode(s));           ❽
               }
70         }
           st.skip("]");
       }
   }
```

- ❶ Main procedure for expanding a compact form.
- ❷ Creates a new empty DOM document.
- ❸ Creates a special purpose tokenizer, described in Example 9.1, to process the file whose name is given as a parameter of the program
- ❹ Gets the identifier before the first opening bracket and uses it as the name for the root element inserted as a child of the document.
- ❺ Expands the rest of the file as a child of the root just created.
- ❻ Creates an *identity* transformer for serializing the output.
- ❼ Makes the serializer pretty-print the output.
- ❽ Creates a transformation source from the DOM structure created.
- ❾ Recursive procedure for expanding the content of a file whose tokens are returned by `st`. The content is inserted as the child of the `elem` element.
- ❿ Finds a reference to the document node necessary for creating new element and text nodes.
- ⓫ Loops on all attribute name and value pairs.
- ⓬ Adds the attribute to the current element.
- ⓭ Loops on all elements until the closing bracket.
- ⓮ Creates a new element with name `s` and adds it as a child of the current element. Recursively expands the next element as a child of the newly created child.
- ⓯ Adds a text node to the current element.

9.2. Creating a Document with SAX Events

Another way of creating an XML document is to have a `Transformer` do it for us. Since the transformer must receive an XML document, we might think that this is pointless. However, a SAX transformer creates a document from SAX events as we saw in Section 8.2. So if we manage to send these types of events while reading our text file, we will obtain a new XML document. This illustrates a clever and efficient way to convert non-XML files into XML.

The main class for the SAX transformation (Example 9.3) is very simple: it creates a `CompactReader` that will read the file as an `InputSource` (line 20-❶); then it creates a transformer to process this source into an output stream (here `System.out`). All the *magic* of setting the input file and creating the document elements is done via the *identity* transformation process (line 25-❷).

Example 9.3. [SAXExpand.java]: XML document creation using SAX events

```
1 import org.xml.sax.InputSource;
  import javax.xml.transform.sax.SAXSource;
  import javax.xml.transform.stream.StreamResult;
  import javax.xml.transform.Transformer;
5 import javax.xml.transform.TransformerFactory;
  import javax.xml.transform.TransformerException;
  import javax.xml.transform.TransformerConfigurationException;

  import java.io.BufferedReader;
10 import java.io.FileReader;
  import java.io.IOException;

  public class SAXExpand {
    public static void main( String[] argv ) {
15      try {
          InputSource inputSource =
              new InputSource(
                  new BufferedReader(new FileReader(argv[0])));
          CompactReader saxReader = new CompactReader();
20          SAXSource source = new SAXSource(saxReader, inputSource); ❶
          StreamResult result = new StreamResult(System.out);

          TransformerFactory tFactory =
              TransformerFactory.newInstance();
25          Transformer transformer = tFactory.newTransformer(); ❷
          transformer.transform(source, result);
        } catch (TransformerException ex){
          System.out.println("TransformerException"+ex);
          ex.printStackTrace();
30      } catch (IOException ex) {
          System.out.println("IOException"+ex);
          ex.printStackTrace();
        }
    }
35 }
```

- ❶ Creates a `Reader` to be used as an `InputSource`.
- ❷ Uses an identity transformer to copy the input to the output stream (here the standard output).

The reading of the file and generation of SAX events are performed by `CompactReader` (Example 9.4), a specialized `XMLReader`. Because `CompactReader` implements the `XMLReader` interface, it must define many methods but only a few of them are really important in this special case. This explains the many stub methods at the end of Example 9.4 (line 89-❸).

The SAX parsing events are sent to an event handler (line 18-❹) for which we define (`get...` and `set...`) accessor methods available to the user of the handler. The main method is `parse` (line 41-❺) that uses the same algorithm as the ones used for the DOM approach. `parse` uses a `CompactTokenizer` instance (line 39-❻) of the same class used with the DOM approach in Example 9.1. This method will generate SAX events as it goes through the text file. It first finds the name of the root element (line 400-❼) and then calls `expand` (line 50-❼) with an indentation level (used only to obtain a nicer output).

Most of the processing is done within the `expand` method that receives the name of the current element as parameter. It first creates an `AttributesImpl` data structure which is populated with the names and values of all attributes (line 64-❾). Once all the attributes have been gathered, we can send a `startElement event` to the handler (line 72-❿). We then process the content of the event either with a recursive call to `expand` (line 77-⓫) or by creating a text element by sending a `characters event` to the handler (line 80-⓬). Once all children elements have been processed, an `endElement event` is sent to the handler (line 85-⓭).

The previous processing produces a valid XML file but one that is very hard to read because everything will appear on the same line. A way to produce a nicely formatted output is to send *ignorable spacing* to the handler. As the formatting rule we use is to always end the current line and add some indentation, we have defined an auxiliary method `ignorableSpacing` that manages an array of characters of the appropriate length. It is initialized with nine spaces but it expands when necessary. This method is called at the appropriate moment in the expansion process, i.e. before creating a new start (line 71-ⓐ) or end element (line 84-ⓑ) or before a new text node (line 79-ⓐ).

Example 9.4. [`CompactReader.java`]: Compact form parsing to generate SAX events

```
1 import org.xml.sax.XMLReader;
  import org.xml.sax.ContentHandler;
  import org.xml.sax.DTDHandler;
  import org.xml.sax.EntityResolver;
5 import org.xml.sax.ErrorHandler;
  import org.xml.sax.InputSource;
  import org.xml.sax.Attributes;
  import org.xml.sax.SAXException;
  import org.xml.sax.helpers.AttributesImpl;
10
  import java.io.IOException;
  import java.io.StreamTokenizer;
  import java.util.Arrays;

15 public class CompactReader implements XMLReader{
    private String nsu = ""; // no namespace URI
    private ContentHandler handler;
```

❶


```

20     private static char[] blanks = "\n          ".toCharArray();           ❷

    private void ignorableSpacing(int nb) throws SAXException {           ❸
        if(nb>blanks.length){// extend the length of space array
            blanks = new char[nb];
25         blanks[0]='\n';
            Arrays.fill(blanks,1,blanks.length,' ');
        }
        handler.ignorableWhitespace(blanks,0,nb);
    }

30     //Return the current content handler.
    public ContentHandler  getHandler(){return handler;}
    //Allow an application to register a content event handler.
    public void setHandler(ContentHandler handler){
35         this.handler=handler;
    }

    //Parse an XML document.
    private CompactTokenizer st;                                         ❹

40     public void parse(InputSource input){                               ❺
        try{
            String rootName = "dummyRoot";
            st = new CompactTokenizer(input.getCharacterStream());
45         // ignore everything preceding the word before the first "["
            while(!st.nextToken().equals("[")){
                rootName=st.getToken();
            }
            handler.startDocument();
50         expand(rootName,1);                                           ❻
            ignorableSpacing(1);
            handler.endDocument();
        } catch (SAXException e){
            System.out.println(e.getMessage());
55     } catch (IOException e) {
            System.out.println("IO Error:"+e);
        }
    }

60     void expand(String elementName,int indent)                         ❽
        throws IOException,SAXException {
        AttributesImpl attrs = new AttributesImpl();
        st.skip("[");
        while(st.getToken().equals("@")) {// process attributes         ❾
65         String attName = st.nextToken();
            st.nextToken();
            attrs.addAttribute(nsu,attName,attName,"CDATA",st.skip("["));
            st.nextToken();
            st.skip("]");
70     }
        ignorableSpacing(indent);                                       ❿
        handler.startElement(nsu,elementName,elementName,attrs);      ⓫
    }

```

```
        while(!st.getToken().equals("]")){ // process content      12
            String s = st.getToken().trim();
75            st.nextToken();
            if(st.getToken().equals("["))
                expand(s,indent+3);      13
            else {
                ignorableSpacing(indent+3);      14
80                handler.characters(s.toCharArray(),0,s.length());      15
            }
        }
        st.skip("]");
        ignorableSpacing(indent);      16
85        handler.endElement(nsu,elementName,elementName);      17
    }

    // dummy definitions...
    public DTDHandler getDTDHandler(){return null;}      18
90    public EntityResolver getEntityResolver(){return null;}
    public ErrorHandler getErrorHandler() {return null;}
    public boolean getFeature(String name) {return false;}
    public Object getProperty(String name) {return null;}
    public void parse(String systemId){}
95    public void setDTDHandler(DTDHandler handler){}
    public void setEntityResolver(EntityResolver resolver){}
    public void setErrorHandler(ErrorHandler handler){}
    public void setFeature(String name, boolean value){}
    public void setProperty(String name, Object value){}
100 }
```

- ❶ Private variable for keeping track of the document handler.
- ❷ Character array containing a carriage return followed by a certain number of blanks that will be used for formatting.
- ❸ Method for returning blanks that will be ignored by the parser.
- ❹ Variable for the tokenizer that handles the compact output.
- ❺ Starting point of the parsing process. Creates the tokenizer for the input file and generates a `startDocument` event.
- ❻ Finds the name of the root node.
- ❼ Starts the recursive processing of the content of an element with the name of the root node.
- ❽ Recursive processing of an element content.
- ❾ Loops over the content of all attributes that are inserted into a single `AttributesImpl` structure.
- ❿ Generate spacing for pretty-printing the output structure.
- ⓫ Generates a `startElement` event with the name of the element and the structure containing the attributes.
- ⓬ Loops over all children nodes in order to process either element or text nodes.
- ⓭ If it is an element node, calls `expand` recursively but with more indentation.
- ⓮ If it is a text node, generates a `characters` event with an appropriate indentation
- ⓯ Indentation before a text node.
- ⓰ Indentation before closing an element
- ⓱ Indicates the end of an element.
- ⓲ Empty definitions for handlers required by the `XMLReader` interface but not used in this example.

9.3. Creating a Document with StAX streaming

To create a new document using the StAX API, we must first create an `XMLStreamWriter` that provides methods to produce XML opening and closing tags, attributes and character content. As this approach only keeps a small part of the document in memory, there is no validation or even well-formedness guarantee on the program. The methods must be called in the appropriate order in order to reflect the XML tree structure.

The most frequently `XMLStreamWriter` (`xmlsw`) methods are:

<code>xmlsw.writeStartDocument();</code>	initialises an empty document to which elements can be added
<code>xmlsw.writeStartElement(String s)</code>	creates a new element named <code>s</code>
<code>xmlsw.writeAttribute(String name, String value)</code>	adds the attribute <code>name</code> with the corresponding <code>value</code> to the last element produced by a call to <code>writeStartElement</code> . It is possible to add attributes as long as no call to <code>writeStartElement</code> , <code>writeCharacters</code> or <code>writeEndElement</code> has been done.
<code>xmlsw.writeEndElement</code>	close the last <i>started</i> element
<code>xmlsw.writeCharacters(String s)</code>	creates a new text node with content <code>s</code> as content of the last started element.

Like we did in Section 9.1, to simplify parsing, we create a customized `StreamTokenizer` (line 18-②) which returns a single token for all characters between separators used in the compact form (i.e. opening and closing brackets, at-sign and newline). The separators are also returned as a single token. The implementation of this tokenizer is given in Example 9.1.

The main class for the StAX transformation (Example 9.5) is very simple: it creates an `XMLStreamWriter` instance that will write the output file (line 17-①) and an instance of a `CompactTokenizer` to read the input file. After skipping everything before the first opening bracket, we initialize the output with a start document (line 26-③) and call `expand` method (line 29-④) to deal with the content of an element. As the output of an `XMLStreamWriter` is not indented, we decided here to define the `ignorableSpacing` method (line 73-⑤) that outputs whitespace nodes. This is usually not needed but it is convenient to be able to see the results. In this particular case, creating a properly indented is a bit tricky because we must output a new line only when the last thing written was the end of an element or an attribute.

The `expand` method (line 43-⑦) first outputs the attributes on the current element (line 46-⑧) and then writes either a new element and calls itself recursively to process its contents (line 57-⑩) or it outputs the character content of the element (line 63-⑪).

Example 9.5. [StAXExpand.java]: XML document creation using the StAX approach

```

1 import java.io.BufferedReader;
  import java.io.FileReader;
  import java.io.IOException;

5 import javax.xml.stream.XMLOutputFactory;
  import javax.xml.stream.XMLStreamException;
  import javax.xml.stream.XMLStreamWriter;

```

```
import java.util.Arrays;
10 public class StAXExpand {
    static XMLStreamWriter xmlsw = null;
    public static void main(String[] argv) {
        try {
15         xmlsw = XMLOutputFactory.newInstance()
                .createXMLStreamWriter(System.out);
            CompactTokenizer tok = new CompactTokenizer(           ❶
                new FileReader(argv[0]));                           ❷

20         String rootName = "dummyRoot";
            // ignore everything preceding the word before the first "["
            while(!tok.nextToken().equals("[")){
                rootName=tok.getToken();
            }
25         // start creating new document
            xmlsw.writeStartDocument();                               ❸
            ignorableSpacing(0);
            xmlsw.writeStartElement(rootName);
            expand(tok,3);                                           ❹
30         ignorableSpacing(0);
            xmlsw.writeEndDocument();                               ❺

            xmlsw.flush();
            xmlsw.close();

35         } catch (XMLStreamException e){                           ❻
            System.out.println(e.getMessage());
        } catch (IOException ex) {
            System.out.println("IOException"+ex);
            ex.printStackTrace();
40         }
    }

    public static void expand(CompactTokenizer tok, int indent)     ❼
        throws IOException,XMLStreamException {
45         tok.skip("[");
            while(tok.getToken().equals("@")) { // add attributes   ❽
                String attName = tok.nextToken();
                tok.nextToken();
                xmlsw.writeAttribute(attName,tok.skip("["));
50         tok.nextToken();
            tok.skip("]");
        }
        boolean lastWasElement=true; // for controlling the output of newlines
        while(!tok.getToken().equals("]")){ // process content     ❾
55         String s = tok.getToken().trim();
            tok.nextToken();
            if(tok.getToken().equals("[")){                          ❿
                if(lastWasElement)ignorableSpacing(indent);
                xmlsw.writeStartElement(s);
60         expand(tok,indent+3);
            lastWasElement=true;
        }
    }
}
```

```

        } else {
            xmlsw.writeCharacters(s);
            lastWasElement=false;
65     }
    }
    tok.skip("\"");
    if(lastWasElement)ignorableSpacing(indent-3);
    xmlsw.writeEndElement();
70 }

private static char[] blanks = "\n".toCharArray();
private static void ignorableSpacing(int nb)
    throws XMLStreamException {
75     if(nb>blanks.length){// extend the length of space array
        blanks = new char[nb+1];
        blanks[0]='\n';
        Arrays.fill(blanks,1,blanks.length, ' ');
    }
80     xmlsw.writeCharacters(blanks, 0, nb+1);
}
}
}

```

- ❶ Creates the `XMLStreamWriter` for outputting the XML text.
- ❷ Creates a specialized tokenizer for the compact form.
- ❸ Initializes the XML document.
- ❹ Calls the expanding process on the root element.
- ❺ Terminates the XML writing proces by ending the document and closing the file.
- ❻ Deals with exceptions that can be raised during writing the XML code and the file.
- ❼ Recursive method to output an element properly indented.
- ❽ Adds attributes name and value on the last element outputted.
- ❾ Deals with the content of the element.
- ❿ Recursively calls the expansion on the content of the element.
- ⓫ Outputs the character content.
- ⓬ Outputs the end of the current element.
- ⓭ Creates a character node containing a new line and an appropriate number of spaces.

9.4. Additional Information on XML Document Creation

Java & XML [36] is a very good information source about creating and manipulating XML documents in Java. It also shows how to integrate stylesheet processing with Java.

All programming languages providing XML parsing APIs, (C [62], C++ [64], C# [63], Perl [42] [40], PHP [77], Python [40], Haskell [50], Prolog [53], Ruby [80] and COBOL [21]) also give means to create XML file by serialization of the DOM-like structures or with SAX callbacks.

Chapter 10. Alternative approaches to XML programming

In the previous sections, XML was used as a formalism for data organization and markup. It was processed either with XML tools or using the Java programming language, which provides a fairly complete set of tools for processing XML documents. Since this tree based approach to data organization is now well accepted, tools for XML file parsing and creation have been developed for other programming languages. XML packages exist for C++, C#, Perl and even Fortran with capabilities and programming interfaces similar to the ones we have presented for Java. XML thus serves as convenient exchange language between programs written in different programming languages.

Of particular interest is `libxml2` library [73] which is an XML parser and toolkit written in C with a variety of language bindings for other programming languages. It is very portable and can be run on most systems. It is most often accessible from the command line thru the `xmllint` command. The parser can validate with DTD, XML Schema, Relax NG or Schematron. XPath and XSLT are also implemented but at the 1.0 level. Some features of XSLT 2.0 have been implemented in a separate project [74].

We will first present how XML data can be processed in Ruby[44], a dynamically typed scripting language known for its flexibility and its large library of tools for creating Web-based applications. We then show how XML can be processed with Python [32], a popular script language, PHP [76], a server-side script language for the Internet, JavaScript [68] used in web browsers, and Swift [85], , a statically typed script language. For mainly historical purposes, we also describe *EcmaScript for XML* (E4X).[66],

We will illustrate the use of XML within these programming languages using the same *compact* and *expand* applications that we have used in previous chapters. This will allow an easier comparison between different programming languages.

Even though we have argued until now that XML is *the ideal* markup language, we will describe in this chapter some alternative approaches for a tree based data organization. It should be remembered that most of these *simplified* XML packages do not provide validation for their input, which must be performed by other means. In the last section of this chapter, we will show other XML *competitors* for exchanging data between programs: *JavaScript Object Notation* (JSON) [71] and *YAML* (YAML Ain't a Markup Language) [90]. These notations are geared toward the production of a human-readable notation for describing tree based informations compared with XML. This goal is achieved but they do not provide any notion of validation or transformation tools within the exchange language itself. But in the context of simple information transfers between programs (e.g. between a browser and a Web server using the AJAX technology for a more pleasant interaction) not needing sophisticated validation or transformation, they provide interesting lightweight alternatives.

10.1. XML processing with Ruby

Ruby [81] is a dynamic object-oriented programming language known for its flexibility and uniformity when accessing object properties. Because XML tree structures can easily be represented with Ruby objects, it is possible to use a simple and intuitive way of dealing with XML objects using essentially the same syntax as that used for other Ruby objects. In Ruby, XML processing is most often performed using the

REXML package. In this section, we will show how to use it for both DOM and SAX parsing, which we used in XML processing in Java.

10.1.1. DOM parsing using Ruby

To show how to process an existing XML structure, we will use the compacting process that we programmed in Java in Section 8.1. In Ruby, DOM parsing is achieved simply by creating an instance of the `REXML::Document` class. This instance is a node and its node type is given by its class: `Element` or `Text` in our example. Information about a node is available through properties `name`, `attributes` (a `Hash` whose keys are the attribute names), `children` (an `Array` that can be iterated upon with `each`) or `value` (more useful for text nodes).

Example 10.1. [DOMCompact.rb] Text compaction of the cellar book (Example 2.2) with Ruby using the DOM model

Compare this program with Example 8.1.

```
1 require 'rexml/document'                               ❶
  include REXML

  def compact(node,indent)                               ❷
5   if(node.class==Element)                             ❸
      print node.name+"["
      indent += ' '*(node.name.length+1);
      first=true;
      node.attributes.each do |key,value|              ❹
10      print "\n#{indent}" unless first
          print "@#{key}[#{value}]"
          first=false;
      end
      node.children.each do |child|                   ❺
15      # deal only with element nodes or non-"empty" text nodes
          if child.class==Element || child.value.strip.length>0 ❻
              print "\n#{indent}" unless first
              compact(child,indent)
              first=false;
20      end
      end
      print "]"
      elsif node.class==Text                          ❼
25      print node.value.strip.gsub(/ *\n */," ") # normalize new lines
  end
  end

  doc = Document.new(STDIN)                            ❽
  compact(doc.root,"")                                 ❾
30
```

- ❶ Makes sure that the `Document` class in the `REXML` library is present. Includes all classes of the `REXML` module.

- ② Prints the content of the XML node. Each line prefixed with an indentation, a string composed of the appropriate number of spaces.
- ③ If the node is an element, prints the name of the node followed by an opening bracket and adds the appropriate number of spaces to the current indentation.
- ④ Deals with attributes which are contained in a hash over which we iterate. For each attribute, prints a @, its key and the corresponding value within square brackets. A new line is started for each attribute except the first.
- ⑤ Processes all children with an iterator.
- ⑥ If it is an element or a non-empty text node, recursively calls `compact` possibly changing line if it not the first child.
- ⑦ Processes a text node by printing it and normalizing internal newlines.
- ⑧ Parses the file by creating a new XML document from the content of the standard input.
- ⑨ Call the compacting process on the document node with an empty indentation string.

10.1.2. SAX parsing using Ruby

In order to process an XML structure with the SAX approach, we will use a similar compacting process to the one we programmed in Java in Section 8.2. SAX parsing is achieved by creating an instance of the `SAX2Parser` class to which we add a listener for parsing events. The parsing process will send *call-backs* to the handler of these events which will produce the compacted document.

Example 10.2. [`SAXCompact.rb`] Text compaction of the cellar book (Example 2.2) with Ruby using the SAX model

Compare this program with Example 8.3.

```

1 require 'rexml/document'                               ❶
  require 'rexml/parsers/sax2parser'
  require 'rexml/sax2listener'
  require 'CompactHandler'
5
  parser = REXML::Parsers::SAX2Parser.new(STDIN)         ❷
  parser.listen(CompactHandler.new)                     ❸
  parser.parse                                           ❹

```

- ❶ Makes sure that the appropriate SAX parser classes of the REXML library and the `CompactHandler` class are present.
- ❷ Creates an instance of the SAX parser that will parse the standard input.
- ❸ Has the parser send its parsing events to the `CompactHandler` class described in Example 10.3.
- ❹ Starts the parsing process.

SAX parsing in Ruby is only a matter of defining the appropriate handlers. Because the current version of REXML does not handle XML entities, some code is added to deal with their declaration and their expansion.

Example 10.3. [`CompactHandler.rb`] Ruby SAX handler for text compacting an XML file such as that of Example 2.2

Compare this program with Example 8.4.

```
1 class CompactHandler
  include REXML::SAX2Listener
  def initialize
    @closed=false
5    @indent=0;
    @entities = {"&"=>"&", "&quot;"=>"'", "&apos;"=>"'",
                 "&lt;" => "<", "&gt;" => ">"}
  end

10 def start_element(uri,localname,qname,attributes)
    if @closed
      print "\n"+" "*@indent
      @closed=false
    end
15    @indent += 1+localname.length()
    print localname+"["
    first=true;
    attributes.each do |key,value|
      print "\n"+" "*@indent if !first
20      first=false
      print "@#{key}[#{expandEntities(value)}]"
      @closed=true
    end
  end
end

25 def end_element(uri,localname,qname)
  print "]"
  @closed=true
  @indent=@indent-1-localname.length
30 end

def characters(text)
  if text.strip.length>0
    if @closed
35      print "\n"+" "*@indent
      @closed=false
    end
    text.strip! # remove leading and trailing space
    text.gsub!(/[s\r\n]+/, ' ') # normalize space
40    print expandEntities(text)
    @closed=true
  end
end

45 def entitydecl(name,decl)
  @entities["&"+name+";"]=decl
end

def expandEntities(text)
50  if text.include?("&")
    # match entities as long as there are replacements
    @entities.each{|key,value| retry if text.gsub!(key,value)}
    # replace numerical entities starting with &#
    while true
```

```

55     if res=text.match("&#x([0-9a-fA-F]+);") # hexadecimal ❶
        text=res.pre_match+[res[1].hex].pack("U")+res.post_match
        elsif res=text.match("&#([0-9]+);") # decimal
            text=res.pre_match+[res[1].to_i].pack("U")+res.post_match
        else
60         break
        end
        end
        end
        end
        text
65     end
    end
end

```

- ❶ Includes the default declarations needed to handle events sent by a SAX parser.
- ❷ Defines and initializes two instance variables needed to output the element with the correct indentation. Creates the entities table and initializes it with the predefined XML entities.
- ❸ When a new element is started, finishes the current indentation if needed.
- ❹ Updates the current indentation by adding the length of the element name.
- ❺ Prints the first attribute on the same line and the others on the subsequent lines properly indented.
- ❻ When an element finishes, closes the current bracket and updates the current indentation.
- ❼ For a non-empty text node, ends current line if needed.
- ❽ Removes leading and trailing space, normalizes and expands entities.
- ❾ When a new entity is declared, add it to the entities table.
- ❿ If the text contains an ampersand, it means that it contains entities the program must deal with. Entities from the entities table are expanded as often as it is necessary.
- ⓫ Numerical entities are expanded by replacing them with their Unicode equivalent.

10.1.3. Creating an XML document using Ruby

In order to demonstrate the creation of new XML document, we will parse the compact form produced in Section 10.1.1 or in Section 10.1.2 like we did in Chapter 9. We first need a way to access appropriate *tokens* corresponding to important signals in the input file. This is achieved by defining a `CompactTokenizer` class (Example 10.4) that will return opening and closing square brackets, at-signs and the rest as a single string. Newlines will also delimit tokens but will be ignored in the document processing. The file is processed by a series of calls to `nextToken` or `skip`. `skip` provides a useful redundancy check for tokens that are encountered in the input but are ignored for output.

Example 10.4. [`CompactTokenizer.rb`] Specialized string scanner that returns tokens of compact form.

Compare this program with Example 9.1.

```

1 require 'strscan' ❶
  class CompactTokenizer ❷
    attr_reader :token ❸
  end

5  def initialize(file) ❹
    @re = /\[|\]|\\n|@[^[^\\]\n@]+/ # handle different token types
    @scanner = StringScanner.new(file.read)
  end
end

```

```
end

10 # get next non-whitespace token and return it
    def nextToken ❶
        loop do
            @token = @scanner.scan(@re)
            break if @token !~ /^\\n? *$/ #skip tokens with only whitespace
15     end
        @token
    end

    # if the current token is "sym" then get next one
20 # otherwise, output error message
    def skip(sym) ❷
        if (@token==sym)
            nextToken
        else
25     # output error message giving the position and the current line
            str = @scanner.string
            pos = @scanner.pos
            raise "skip:#{sym} expected but #{@token} found at position #{pos}:" +
                str[str.rindex("\\n",pos)||0 ... str.index("\\n",pos)||str.length]
30     end
        end
    end
end
```

- ❶ Includes the `StringScanner` package on which this tokenizer is built.
- ❷ Defines the `CompactTokenizer` class.
- ❸ Makes the current token available as a readable attribute.
- ❹ `@re` is an instance variable containing the regular expression corresponding to each token type; `@scanner` is the `StringScanner` instance running on the string read from the `fileName` parameter.
- ❺ Gets the next token but skip those containing only whitespace. It returns the value of the current token.
- ❻ Checks that the current token is the same as the one given as parameter and retrieves the next one. If the current token does not match, then raises an exception giving the expected token and the one found with the content of the current line.

Thanks to the dynamic object oriented aspect of Ruby, much of the complexity of node creation and child addition is hidden in simple class instance creations: `Document.new()`, `Element.new()` and `Text.new()`. Adding a new node is done with the `<<` operator that appends new information at the end of almost any object in Ruby. The attributes of a node are kept in a hash table that is a property of an XML node. This hash table is assigned like any other one in Ruby.

Example 10.5. [DOMExpand.rb] Ruby compact form parsing to create an XML document

A sample input for this program is Example 5.8 , which should yield Example 2.2. Compare this program with Example 9.2.

```

1 require 'rexml/document' ❶
  include REXML

  require 'compactTokenizer' ❷
5 st = CompactTokenizer.new(STDIN) ❸

  # start new document
  doc = Document.new() ❹
  doc << XMLDecl.new(1.0,"UTF-8") ❺
10 # find the name of the root
    rootName = 'dummyElement' ❻
    while st.nextToken!='['
      rootName=st.token
    end

15 # create the element and return it with the current token
    def expand(st,elementName) ❼
      elem = Element.new(elementName) ❽
      st.skip("[")
20 while st.token=='@' # process attributes ❾
      attName = st.skip("@")
      st.nextToken
      elem.attributes[attName] = st.skip("[") ❿
      st.nextToken
25 st.skip("]")
    end
    while st.token!=']' # process children ⓫
      s = st.token.strip
      st.nextToken
30 if(st.token=='[') ⓬
      elem << expand(st,s)
    else ⓭
      elem << Text.new(s) ⓮
    end
35 end
    st.skip("]")
    return elem
  end

40 # expand from the root element
  doc << expand(st,rootName) ⓯
  # write it properly indented
  doc.write(STDOUT,0) ⓰

```

- ❶ Ensures that the library REXML document class is loaded, then include it.
- ❷ Ensures that the CompactTokenizer (Example 10.4) is loaded.
- ❸ Creates the tokenizer on the standard input.
- ❹ Initializes a new XML document.
- ❺ Adds an appropriate XML declaration.
- ❻ The name of the root is the first token immediately followed by an opening square bracket.
- ❼ Creates an XML element corresponding to the content of `rootName` and returns it.
- ❽ Creates a new, empty XML element with name `elementName`.

- ⑨ Because all attribute names start with an @, we loop while the current token is equal to @. The name of the attribute is saved and the following opening square bracket is skipped, the value is kept and the] is skipped.
- ⑩ Adding an attribute is done by assigning to the `attributes` hash within the XML node.
- ⑪ All children are then processed in turn until a closing square bracket is encountered. `s` is the current token, which is either a child element name (if followed by an opening square bracket) or the content of a text node.
- ⑫ A child node is expanded by a recursive call whose result is added as the last child of the current element.
- ⑬ A text node is added as the last child of the current element.
- ⑭ Calls the expansion of the document starting from the `rootName` and adds the result as the child of the document.
- ⑮ Serializes the document on the standard output. By default, the XML is properly indented. The second parameter gives a global indentation level if needed.

10.2. XML processing with Python

Python [32] is a dynamically typed script programming language with a syntax inspired by functional languages such as Haskell. In this section, we will show how to use it for both DOM, SAX and StAX parsing, which we used in XML processing in Java. As most XML files use the UTF-8 encoding, we will use Version 3 of Python in which the processing of this encoding is better integrated. To make string processing more uniform, it preferable to also use UTF-8 as the encoding of our Python source file. This is indicated using a structured comment as the first line of our programs (see Example 10.6).

10.2.1. DOM parsing using Python

To show how to process an existing XML structure, we will use the compacting process that we programmed in Java in Section 8.1. In Python, DOM parsing is achieved simply by calling the `parse` method from the `xml.dom.minidom` package. This creates a DOM structure composed of instances of the `xml.dom.Node` class.

Example 10.6. [DOMCompact.py] Text compaction of the cellar book (Example 2.2) with Python using the DOM model

Compare this program with Example 8.1.

```

1 # -*- coding: utf-8 -*-
  import xml.dom
  from xml.dom import Node
  from xml.dom.minidom import parse
5 import sys

  def strip_space(node):
    child=node.firstChild
    while child!=None:
10         c=child.nextSibling
            if (child.nodeType==Node.TEXT_NODE and
                len(child.nodeValue.strip())==0):
                node.removeChild(child)
            else:
15                 strip_space(child)
                child=c
    return node

# use sys.stdout.write instead of print to control the output
20 # i.e. without added spaces between elements to print
  def compact(node,indent):
    if node==None: return
    if node.nodeType==Node.ELEMENT_NODE:
        sys.stdout.write(node.nodeName+' ')
        indent += (len(node.nodeName)+1)*" "
        attrs = node.attributes
        first=True
25         for i in range(len(attrs)):
            if not first: sys.stdout.write('\n'+indent)

```

```
30         sys.stdout.write('@'+attrs.item(i).nodeName +
                           '['+attrs.item(i).nodeValue+']')
        first=False
        child=node.firstChild
        while child!=None:
35             if not first: sys.stdout.write('\n'+indent)
                compact(child,indent)
                first=False
                child=child.nextSibling
            sys.stdout.write(']')
40     elif node.nodeType==Node.TEXT_NODE:
        sys.stdout.write(node.nodeValue.strip())

    doc = parse(sys.stdin)
    compact(strip_space(doc.documentElement), "")
45 sys.stdout.write('\n')
```

- ❶ Imports the `xml.dom` module from which we specifically import the `Node` class and the `parse` method to read the standard input.
- ❷ A recursive method which goes through the DOM structure to remove whitespace only nodes. It returns the *cleaned* DOM structure.
- ❸ Prints the content of the XML node. Each line is prefixed with an indentation, a string composed of the appropriate number of spaces.
- ❹ If the node is an element, prints the name of the node followed by an opening bracket and adds the appropriate number of spaces to the current indentation.
- ❺ Deals with attributes which are contained in a list of nodes over which we iterate. For each attribute, prints a @, its name and the corresponding value within square brackets. A new line is started for each attribute except the first.
- ❻ Loops over the list of children possibly changing line if it not the first child.
- ❼ Recursively calls `compact` on the child node with the updated indentation.
- ❽ Processes a text node by printing it and normalizing internal newlines.
- ❾ Parses the file by creating a new XML document from the content of the standard input.
- ❿ Calls the compacting process on the document node with an empty indentation string and ends the last line with a newline.

10.2.2. SAX parsing using Python

In order to process an XML structure with the SAX approach, we will use a similar compacting process to the one we programmed in Java in Section 8.2. SAX parsing is achieved by creating an instance of the `SAX` class to which we add a handler method for the corresponding parsing events. The parsing process will send *call-backs* to the handler of these events which will produce the compacted document.

Example 10.7. [`SAXCompact.py`] Text compaction of the cellar book (Example 2.2) with Python using the SAX model

Compare this program with Example 8.3.


```

1 import xml.sax                                     ❶
  import sys
  import CompactHandler

5 parser = xml.sax.make_parser()                     ❷
  parser.setContentHandler(CompactHandler.CompactHandler()) ❸
  parser.parse(sys.stdin)                            ❹
  sys.stdout.write('\n')

10

```

- ❶ Imports the SAX parser package and the `CompactHandler` class.
- ❷ Creates an instance of the SAX parser.
- ❸ Makes the parser send its parsing events to the `CompactHandler` class described in Example 10.8.
- ❹ Starts the parsing process and ends the last line with a newline.

SAX parsing in Python is only a matter of defining the appropriate handlers. Because nodes are not normalized, many successive text nodes can appear within an element, so some care has to be taken to deal with this fact.

Example 10.8. [`CompactHandler.py`] Python SAX handler for text compacting an XML file such as that of Example 2.2

Compare this program with Example 8.4.

```

1 from xml.sax.handler import ContentHandler         ❶
  import sys

  class CompactHandler(ContentHandler):             ❷
5     def __init__(self):
      self.closed = False
      self.indent = 0
      self.lastNodeWasText = False

10    def startElement(self, localname, attrs):      ❸
      if self.closed:
          sys.stdout.write('\n'+self.indent*" ")
          self.closed = False
          self.indent+=1+len(localname)             ❹
15    sys.stdout.write(localname+'[')
      first = True
      for name in attrs.getNames():                 ❺
          if not first:
              sys.stdout.write('\n'+self.indent*" ")
20    first = False
          sys.stdout.write('@'+name+'['+attrs.getValue(name)+'']')
          self.closed = True
          self.lastNodeWasText = False

25    def endElement(self, localname):              ❻
      self.closed = True
      sys.stdout.write(']')

```

```
        self.indent -= 1+len(localname)
        self.lastNodeWasText = False
30
    def characters(self, data):
        data = data.strip()
        if(len(data)>0):
            if self.closed and not self.lastNodeWasText:
35                sys.stdout.write('\n'+self.indent*" ")
                self.closed = False
                sys.stdout.write(data)
                self.closed=True
            self.lastNodeWasText = True
```

- ❶ Imports the `xml.sax.handler` package containing the `ContentHandler` class which will be subclassed to handle events sent by a SAX parser.
- ❷ Defines the constructor of the class and initializes three instance variables needed to output the element with the correct indentation. The last flag is used when many successive text nodes appears; this happens when XML entities are being replaced by their values.
- ❸ When a new element is started, finishes the current indentation if needed.
- ❹ Updates the current indentation by adding the length of the element name.
- ❺ Prints the first attribute on the same line and the others on the subsequent lines properly indented.
- ❻ When an element finishes, closes the current bracket and updates the current indentation.
- ❼ For a non-empty text node, ends current line if needed and if the last node was not a text node. Writes the content of the node and indicates that the last node was a text node.

10.2.3. StAX parsing using Python

Pull parsing in Python is performed using the `pullDOM` package whose interface to the `DOMEventStream` is a bit tricky to use. In particular, it can return many successive text nodes and it does not provide an uniform API like the one defined in the Java `XMLStreamReader` class. So to simplify our processing, we first define our own `XMLStreamReader` class which will be used in Example 10.10.

Example 10.9. [XMLStreamReader.py] Text compaction of the cellar book (Example 2.2) with Python using the StAX model

```
1 from xml.dom import pullDOM
  import sys
                                     ❶

class XMLStreamReader():
5     """
    XMLStreamReader: simplified interface to DOMEventStream
    so that it looks similar to the Java XMLStreamReader with similar methods
    It concatenates all successive text nodes

10 """
    def __init__(self, file):
                                     ❷
        self.events = pullDOM.parse(file)
        self.event = None
        self.node = None
15        self.look_ahead=None
                                     ❸
```

```

# get next token
def next(self):
    if self.look_ahead == None:
        (self.event, self.node) = self.events.getEvent()
    else:
        (self.event, self.node) = self.look_ahead
        self.look_ahead = None
    # concatenate consecutive character nodes
    while self.event == pulldom.CHARACTERS:
        self.look_ahead = self.events.__next__()
        while self.look_ahead[0] == pulldom.CHARACTERS:
            self.look_ahead = self.events.getEvent()
        break

def isWhiteSpace(self):
    return (self.event == pulldom.CHARACTERS and
            len(self.node.data.strip()) == 0)

def isStartElement(self):
    return self.event == pulldom.START_ELEMENT

def isEndElement(self):
    return self.event == pulldom.END_ELEMENT

def isCharacters(self):
    return self.event == pulldom.CHARACTERS

def checkStartElement(self, method):
    if not self.isStartElement():
        raise ValueError("%s called for an event of type: %s" %
                          (method, self.event))

def getLocalName(self):
    self.checkStartElement("getLocalName")
    return self.node.localName

def getAttributeCount(self):
    self.checkStartElement("getAttributeCount")
    return self.node.attributes.length

def getAttributeLocalName(self, i):
    self.checkStartElement("getAttributeLocalName")
    return self.node.attributes.item(i).localName

def getAttributeValue(self, i):
    self.checkStartElement("getAttributeValue")
    return self.node.attributes.item(i).value

def getText(self):
    if not self.isCharacters():
        raise ValueError("getText called for an event of type %s" %
                          self.event)
    return self.node.data

```

- ❶ Imports the `pullDOM` package.
- ❷ Defines a new class to simplify access to the pull parsing process.
- ❸ Constructor that defines the instance variables: access to the file, the current event and node. A look-ahead node event pair useful to concatenate the text content of successive text nodes.
- ❹ Gets the next event either from stream if there is no look-ahead or from the look-ahead.
- ❺ Concatenates successive character nodes. The first *non-text* node is kept in the look-ahead for the next call.
- ❻ A series of node type tests on the current event.
- ❼ Checks if the current element is a start node, otherwise raises an exception. Useful for debugging the following methods.
- ❽ Returns information about the current start element.
- ❾ Checks if the current node is a text node and returns its contents if it is the case, otherwise raises an exception.

Example 10.10. [StAXCompact.py] Text compaction of the cellar book (Example 2.2) with Python using the StAX model

Compare this program with Example 8.5.

```
1 from XMLStreamReader import XMLStreamReader
  import sys
  ❶

def compact(xmlsr,indent):
  ❷
5   if xmlsr.isStartElement():
      sys.stdout.write(xmlsr.getLocalName()+'[ ' )
      indent += (len(xmlsr.getLocalName()+1)*" " )
      count = xmlsr.getAttributeCount()
      for i in range(count):
10          if i>0:
              sys.stdout.write(indent)
              sys.stdout.write('@'+xmlsr.getAttributeLocalName(i)+
                  '['+xmlsr.getAttributeValue(i)+' ]')

15          first = count==0
              while True:
                  xmlsr.next()
                  while xmlsr.isWhiteSpace():xmlsr.next()
                  if xmlsr.isEndElement():break;
                  if first:
20                      first=False
                  else:
                      sys.stdout.write(indent)
                      compact(xmlsr,indent);
                      sys.stdout.write(']')
25          elif xmlsr.isCharacters():
              sys.stdout.write(xmlsr.getText().strip())

  xmlsr = XMLStreamReader(sys.stdin)
30 xmlsr.next()
  while not xmlsr.isStartElement():xmlsr.next()
  ❸
  ❹
  ❺
  ❻
  ❼
  ❽
  ❾
```

```
compact(xmlsr, "\n")
sys.stdout.write("\n")
```

- ❶ Imports the XMLStreamReader class defined in Example 10.9
- ❷ If it is a start element tag, outputs the name of the element followed by an opening bracket and update the current indentation.
- ❸ Outputs each attribute name and value properly indented except for the first one. Attributes are obtained by indexing within the loop on the number of attributes.
- ❹ Loops on children nodes that are not whitespace and compacts each of them with the correct indentation.
- ❺ Recursive call to the compacting process.
- ❻ Prints the normalized character content.
- ❼ Creates a new stream reader, indicates that it will parse the standard input.
- ❽ Ignores the tokens that come before the first element (e.g. processing instructions).
- ❾ Calls the compacting process and then prints a newline to flush the content of the last line.

10.2.4. Creating an XML document using Python

In order to demonstrate the creation of new XML document, we will parse the compact form produced in Section 10.2.1 or in Section 10.2.2 like we did in Chapter 9. We first need a way to access appropriate *tokens* corresponding to important signals in the input file. This is achieved by defining a `CompactTokenizer` class (Example 10.11) that will return opening and closing square brackets, at-signs and the rest as a single string. Newlines will also delimit tokens but will be ignored in the document processing. The file is processed by a series of calls to `nextToken` or `skip`. `skip` provides a useful redundancy check for tokens that are encountered in the input but are ignored for output.

Example 10.11. [`CompactTokenizer.py`] Specialized string scanner that returns tokens of compact form.

Compare this program with Example 9.1.

```
1 import re, sys
class CompactTokenizer():
5     def __init__(self, file):
        self.pat = re.compile('[\[\@\]]')
        self.whitespacepat = re.compile('\n? *$')
        self.file = file
        self.tokens = []
10    def nextToken(self):
        while True:
            if len(self.tokens)==0:
                self.line = self.file.readline().strip()
15                if self.line:
                    self.tokens = [tok for tok in
                                self.pat.split(self.line) if tok!='']
            else:
```

```
                self.token = None
20                break
                self.token = self.tokens.pop(0)
                if not self.whitespacepat.match(self.token):
                    return self.token

25    def getToken(self):
        return self.token

        def skip(self, sym):
30            if self.token==sym:
                return self.nextToken()
            else:
                raise ValueError('skip:%s expected but %s found'%(sym,self.token))
```

- ❶ Includes the regular expression `re` package on which this tokenizer is built.
- ❷ Defines the `CompactTokenizer` class.
- ❸ Defines the `pat` regular expression to split an input line on an ampersand, an opening or closing square bracket. The square brackets characters in the regular expression must be preceded by a backslash as square brackets are part of the regular expression language, also used in the same expression. Defines a regular expression to match a whitespace node. Save the reference to the file to tokenize and initializes the current list of tokens.
- ❹ Gets the next token but skip those containing only whitespace. When the current list of tokens is empty, it reads the next line of the file and initializes the list of tokens for this line. It returns `None` at the end of the file. When the list of tokens is not empty, sets the current token to the first one in the list and removes it from the list. If the current token is not a whitespace node, returns it.
- ❺ List comprehension expression to split the current line at separators which are also returned as tokens. Should empty tokens appear, they are removed from the result.
- ❻ Returns the current token.
- ❼ Checks that the current token is the same as the one given as parameter and retrieves the next one. If the current token does not match, then raises an exception giving the expected token and the one found.

In Python, a new document is created using the constructor of the `Document` class. The result of calling the constructor is then assigned to a variable which can be used to create an element or a text node. Adding a new node is done with the `appendChild` method that adds the new node as the last child of a given node. The attributes of a node are added using the `setAttribute` method.

Example 10.12. [DOMExpand.py] Python compact form parsing to create an XML document

A sample input for this program is Example 5.8 , which should yield Example 2.2. Compare this program with Example 9.2.

```
1 import xml.dom.minidom, sys
  from CompactTokenizer import CompactTokenizer

  def expand(elem):
5     global ct,doc
     ct.skip("[")
     while ct.getToken()=='@':
7         attName = ct.nextToken()
```

```

    ct.nextToken()
10     elem.setAttribute(attName,ct.skip("[ "))           ❸
        ct.nextToken()
        ct.skip("]")
    while ct.getToken()!=']':                             ❹
        s=ct.getToken().strip()
15     ct.nextToken()
        if ct.getToken()=="[" :
            expand(elem.appendChild(doc.createElement(s.strip())))  ❺
        else:
            elem.appendChild(doc.createTextNode(s))           ❻
20     ct.skip("]")
    return elem

doc = xml.dom.minidom.Document()                         ❾
ct = CompactTokenizer(sys.stdin)                          ❿
25 while ct.nextToken()!='[' :
    rootName=ct.getToken()
    expand(doc.appendChild(doc.createElement(rootName.strip())))

print (doc.toprettyxml(" "))                             ❿
30

```

- ❶ Imports the necessary packages for creating DOM nodes and for accessing the standard input.
- ❷ Imports the `CompactTokenizer` (Example 10.11).
- ❸ Creates an XML element corresponding to the content of `elem` and returns it.
- ❹ Because all attribute names start with an `@`, we loop while the current token is equal to `@`. The name of the attribute is saved and the following opening square bracket is skipped, the value is kept and the `]` is skipped.
- ❺ Adding an attribute is done by calling the `setAttribute` method with the name of the attribute and its corresponding value.
- ❻ All children are then processed in turn until a closing square bracket is encountered. `s` is the current token, which is either a child element name (if followed by an opening square bracket) or the content of a text node.
- ❼ A child node is expanded by a recursive call whose result is added as the last child of the current element.
- ❽ A text node is added as the last child of the current element.
- ❾ Initializes a new XML document.
- ❿ Creates the tokenizer on the standard input.
- ❿ The name of the root is the first token immediately followed by an opening square bracket. The root node is created as the child of the document node which is then filled by the initial call to `expand`.
- ❿ The resulting DOM node in `doc` is then formatted in indented form to the output using the `toprettyxml` method.

10.2.5. Other means of dealing with XML documents using Python

Python also allows other ways of dealing with XML files. For simple reading and manipulation of information of an XML file, one might consider the `ElementTree` package which, upon parsing an XML file, translates its the tree structure into a Python object, an instance of the `ElementTree` class, that can be processed

using the usual property selectors and array iterators. Methods are provided to get the name of an element, its attributes and its children nodes. In an `ElementTree` instance, access to the children nodes is achieved by iterating on the node. Text content is obtained by reading the `text` property. Mixed content is dealt with the `tail` property of a child node which gives the text content in the document before the next child.

In order to illustrate the manipulation of SimpleXML structures, we give in Example 10.13 a version to produce a compact version of an XML file. We parse (load) the file and the `ElementTree` structure is built in memory. It is then a simple matter of traversing this structure to produce a compact version of the file.

Example 10.13. [ETCompact.py] Python compaction of an XML file using ElementTree nodes.

Compaction of an XML file by first creating an `ElementTree` object and then traversing it with standard Python iterators to create a compact version of the file.

```
1 import xml.etree.ElementTree as ET           ❶
  import sys, re

def stripNS(tag):                               ❷
5     return re.sub("^\{.\+}", "", tag)

def compact(node, indent):                       ❸
    if node==None: return
    if ET.iselement(node):
10     localname = stripNS(node.tag)             ❹
        sys.stdout.write(localname+' ')
        indent += (len(localname)+1)*" "
        attrs = node.attrib
        first=True
15     for name in attrs:                         ❺
            if not first: sys.stdout.write(indent)
            sys.stdout.write('@%s[%s]'%(name, attrs.get(name)))
            first=False
        if node.text and len(node.text.strip())>0:  ❻
20     sys.stdout.write(node.text.strip())
        first=False
        for child in list(node):                 ❼
            if not first: sys.stdout.write(indent)
            compact(child, indent)               ❽
25     first=False
            if child.tail and len(child.tail.strip())>0:
                sys.stdout.write(indent+child.tail.strip())
            sys.stdout.write(' ')

30 compact(ET.parse(sys.stdin).getroot(), "\n")  ❾
    sys.stdout.write('\n')
```

- ❶ Imports the `ElementTree` class and renames it as `ET` because it will be used often. Imports other necessary packages.

- ② As `ElementTree` nodes contain namespace information in their print names, we discard it here to keep only the local name.
- ③ Function to compact a node (first parameter) with each new line preceded by an indentation given by the second parameter.
- ④ Prints the name of the element followed by an open bracket and updates the indentation by adding spaces corresponding to the number of characters in the element name.
- ⑤ Prints the attributes on different lines except for the first. The name of the attribute is preceded by `@` and the value is put within square brackets.
- ⑥ If there is text content, prints it. The text is normalized by removing spaces at the start and end.
- ⑦ If the element has children, compacts them, a new line is output if it is not the first child.
- ⑧ Calls `compact` recursively.
- ⑨ Parses the standard input to get the `ElementTree` root node and calls the compacting process on it. Ends the output with a new line.

Example 10.14 follows the same structure as Example 10.12 to expand a compact form into a `ElementTree` structure. It uses the tokenizer of Example 10.11 to read the file. It recursively creates the XML structure as it processes the file. As the Python library does not provide an indented form of the XML string, we provide one here as another illustration of `ElementTree` use.

Example 10.14. [ETExpand.py] Python compact form parsing to create a `ElementTree` document

A sample input for this program is Example 5.8 , which should yield a file equivalent to Example 2.2. Compare this program with Example 9.2.

```

1 import xml.etree.ElementTree as ET                                ①
  import sys,re
  from CompactTokenizer import CompactTokenizer                    ②

5 def expand(elem):
    global ct
    ct.skip("[")
    while ct.getToken()=='@':                                     ③
        attName = ct.nextToken()
10         ct.nextToken()
           elem.attrib[attName]=ct.skip("[")                       ④
           ct.nextToken()
           ct.skip("]")
    lastNodeWasText=True
15     while ct.getToken()!=']':                                    ⑤
        s=ct.getToken().strip()
        ct.nextToken()
        if ct.getToken()=="[" :                                    ⑥
            child=ET.Element(s)
20             elem.append(child)
                expand(child)
                lastNodeWasText=False
        else:
            if lastNodeWasText:                                    ⑦
25                 if not elem.text:
                    elem.text=""

```

```

        elem.text+=s
    else:
        if not child.tail:
30         child.tail=""
            child.tail+=s
        ct.skip("]")

    ct = CompactTokenizer(sys.stdin)
35 while ct.nextToken()!='[':
        rootName=ct.getToken()
        doc=ET.Element(rootName.strip())
        expand(doc)

40 # ElementTree does not provide a pretty-print method so we define one
    def pprint(elem,indent):
        sys.stdout.write(indent+'<'+elem.tag)
        for a in elem.attrib:
            sys.stdout.write(' %s="%s"'%(a,elem.attrib[a]))
45     newindent = indent+"    "
        sys.stdout.write('>')
        if elem.text:
            sys.stdout.write(elem.text)
        for child in elem:
50         pprint(child,newindent)
            if child.tail:
                sys.stdout.write(newindent+child.tail)
        if elem:
            sys.stdout.write(indent)
55     sys.stdout.write('</'+elem.tag+'>')

pprint(doc,"\n")
sys.stdout.write("\n")

```

- ❶ Imports the `ElementTree`, other system packages and the `CompactTokenizer` (Example 10.11).
- ❷ Adds the content of the file corresponding to the children of the current node to the `elem` element.
- ❸ Because all attribute names start with an `@`, loops while the current token is equal to `@`. The name of the attribute is saved and the following opening square bracket is skipped, the value is kept and the `]` is skipped.
- ❹ Adding an attribute is done by setting the attribute value associated with its name in the `attrib` dictionary of the current element.
- ❺ All children are then processed in turn until a closing square bracket is encountered. `s` is the current token, which is either a child element name (if followed by an opening square bracket) or the content of a text node.
- ❻ A new element named `s` is created and its children are filled in by a recursive call to `expand`.
- ❼ A text node is added as the child of the current element. Successive text nodes are added by concatenating the content of the text node with the content of the last child tail.
- ❽ Creates the tokenizer on the standard input.
- ❾ The name of the root is the first token immediately followed by an opening square bracket.
- ❿ Creates the root node as an `ElementTree`.
- ⓫ The root node which is filled in by a call to `expand`.
- ⓬ Recursive function to create and indented string for the current element. The `indent` parameter is a string which is output before the start of each new line.

- ⑬ Outputs the name of element followed by its attributes on a single line.
- ⑭ If there is a text, outputs it.
- ⑮ Recursively process each children possibly writing the text content of the `tail` property at the same level of indentation. Finally closes the tag.
- ⑯ Prints the document node on the standard output by calling the `pprint` method defined above.

10.3. XML processing with PHP

PHP [76] is a popular server-side script language embedded in HTML that provides excellent string processing capabilities coupled with a good integration with a relational data-base most often MySQL, but other databases can also be accomodated. It features also many libraries for reading, creating and transforming XML data [77]. In this section, we will show how to use PHP for DOM, SAX and StAX parsing. We will use the same algorithms and program organization that we used in Java (Chapter 8 and Chapter 9). We will also describe how to transform XML data using XSL stylesheets and briefly show how to use the SimpleXML library that is useful for some simple cases.

10.3.1. DOM parsing using PHP

To show how to process an existing XML structure, we will use the compacting process that we programmed in Java in Section 8.1. In PHP, DOM parsing is achieved simply by creating an instance of the Dom class. This instance is a node and its node type is given by the value of its property `nodeType`: `XML_ELEMENT_NODE` or `XML_TEXT_NODE`. Information about a node is available through properties `nodeName`, `attributes` (an array whose keys are attribute names), `childNodes` (an array that can be iterated upon with `foreach`) or `textContent` for text nodes.

Example 10.15. [DOMCompact.php] Text compaction of the cellar book (Example 2.2) with PHP using the DOM model

Compare this program with Example 8.1.

```

1 <?php
  function compact_($node,$indent){
    if ($node->nodeType==XML_ELEMENT_NODE) {
      print $node->nodeName."[";
5      $indent.=str_repeat(' ',strlen($node->nodeName)+1);
      $first=true;
      foreach ($node->attributes as $attrName => $attrValue) {
        if(!$first)print "\n$indent";
        print "@".$attrName."[".$attrValue->value."]";
10      $first=false;
      }
      foreach ($node->childNodes as $child)
        if($child->nodeType==XML_ELEMENT_NODE or
          strlen($child->textContent)>0 ){
15      if(!$first)print "\n$indent";
        compact_($child,$indent);
        $first=false;
      }
      print "]";
20    } else if ($node->nodeType==XML_TEXT_NODE)
      print preg_replace("/ *\n? +/", " ",trim($node->textContent));
  }

  $dom = new DomDocument();
25 $dom->preserveWhiteSpace = false;
  $dom->substituteEntities = true;

```

```

$dom->load("php://stdin");
compact_($dom->documentElement, "");
print "\n";
30 ?>

```

- ❶ Prints the content of the XML node. Each line prefixed with an indentation, a string composed of the appropriate number of spaces.
- ❷ If the node is an element, prints the name of the node followed by an opening bracket and adds the appropriate number of spaces to the current indentation.
- ❸ Deals with attributes which are contained in an array over which we iterate. For each attribute, prints a @, its key and the corresponding value within square brackets. A new line is started for each attribute except the first.
- ❹ Processes all children with a `for-each` loop.
- ❺ If it is an element or a non-empty text node, recursively calls `compact` possibly changing line if it not the first child.
- ❻ Processes a text node by printing it and normalizing internal newlines.
- ❼ Parses the file by creating a new XML document from the content of the standard input. Sets parameters to that white space only nodes are not returned and that entities are replaced by their values before the XML processing.
- ❽ Call the compacting process on the document node with an empty indentation string.

Because PHP programs are meant to be embedded in HTML, we now show how the HTML compacting process can be done.

Example 10.16. [`compactHTML.php`] HTML compaction of the cellar book (Example 2.2) with PHP using the DOM model

Compare this program with Example 5.7.

```

1 <?php
  function tag($tag,$body,$attrs=""){
    return "<$tag".($attrs?" ":"")."$attrs>$body</$tag>";
  }
5
  function compact_($node){
    if ($node->nodeType==XML_ELEMENT_NODE) {
      $res="";
      $attrs="";
10      foreach ($node->attributes as $key => $value)
        $attrs.=" $key='$value->value' ";
      $res.=tag("b",$node->nodeName)." ".$attrs;
      $children = $node->childNodes;
      if($children->length>1){
15        $cs="";
        foreach($children as $child)
          $cs.=compact_($child);
        $res.=tag("ul",$cs);
      } else
20        $res.=compact_($children->item(0));
      return tag("li",$res);
    } else if ($node->nodeType==XML_TEXT_NODE){

```

```
        return preg_replace("/ *\\n? +/", " ", trim($node->nodeValue)); ❸
    }
25 }

    $dom = new DomDocument(); ❹
    $dom->preserveWhiteSpace = false;
    $dom->substituteEntities = true;
30 $file = $_GET['file'];
    $dom->load($file?$file:"php://stdin");
    ?>
    <html xmlns="http://www.w3.org/1999/xhtml">
        <head>
35         <title>HTML compaction of "<?php print $_GET['file']?>"</title>
        </head>
        <body>
            <ul>
                <?php print compact_($dom->documentElement, "")?> ❺
40            </ul>
        </body>
    </html>
```

- ❶ Defines a function that returns a string comprised of the string of the body of a tag (second parameter) wrapped by the start and end tag named with the first parameter. The third argument give a string of attributes that are added to the start tag.
- ❷ Returns a string corresponding to the content of the XML node.
- ❸ If the node is an element, the name of the node is and its attributes are embedded in a `b` tag. The content of the attributes and the children nodes is embedded in a `li` tag.
- ❹ Deals with attributes which are contained in an array over which we iterate. For each attribute, prints its name and the corresponding value separated by an equal sign. These attributes are combined in a single string which is added after the name of the node wrapped with a `b` tag.
- ❺ Gets the list of all children.
- ❻ If there are more than one child, accumulates the content of the children and wrap it within an `ul` tag.
- ❼ In the case of one child, returns its content.
- ❽ Processes a text node by normalizing its content.
- ❾ Parses the file by creating a new XML document from the content of a file named by the HTTP file parameter passed in the url. If this value is not set, the standard input content will be used. Sets parameters to that white space only nodes are not returned and that entities are replaced by their values before the XML processing.
- ❿ Calls the compacting process on the document node within an HTML structure forming the shape of HTML page. `compact_` returns a string corresponding to the content of the file which is printed within the HTML template.

10.3.2. SAX parsing using PHP

In order to process an XML structure with the SAX approach, we will use a similar compacting process to the one we programmed in Java in Section 8.2. SAX parsing is achieved by calling the `xml_parser_create` function. This call return a reference to a parser to which we can set some options and then give an object in which are defined *call-back* functions that will be called during the parsing process.

Example 10.17. [SAXCompact.php] Text compaction of the cellar book (Example 2.2) with PHP using the SAX model

Compare this program with Example 8.3.

```

1 <?php
  require 'CompactHandler.php';                                ❶
  $handler = new CompactHandler();

5 $parser = xml_parser_create("UTF-8");                       ❷
  xml_parser_set_option($parser, XML_OPTION_TARGET_ENCODING, "UTF-8"); ❸
  xml_parser_set_option($parser, XML_OPTION_CASE_FOLDING, 0);
  xml_parser_set_option($parser, XML_OPTION_SKIP_WHITE, 1);

10 xml_set_object($parser,$handler);                          ❹
  xml_set_element_handler($parser, "startElement", "endElement");
  xml_set_character_data_handler($parser,"characters");

  $fp = fopen("php://stdin","r");                              ❺
15 while ($data = fread($fp, 4096)) {
    if (!xml_parse($parser, $data, feof($fp))) {
        die(sprintf("XML error: %s at line %d",
                    xml_error_string(xml_get_error_code($parser)),
                    xml_get_current_line_number($parser)));
20     }
    }
  print "\n";
  xml_parser_free($parser);
?>

```

- ❶ Makes sure that the appropriate `CompactHandler` class is present.
- ❷ Creates an instance of the SAX parser that will parse an UTF-8 input. Set some options, in particular, make sure that tag names are not returned in upper-case.
- ❹ Indicates to which object the parsing events of the `CompactHandler` class (described in Example 10.18) will be sent. Then gives the name of functions of this object that will be called when a start tag, an end tag and a character node will be parsed.
- ❺ Starts the parsing process on the standard input. In order to save space, the parsing is sent to the parser in chunks of 4K until the the third parameter of `xml_parse` is `true`.

SAX parsing events in PHP are sent to an event handler object in which are defined some methods to deal with the parsing events. Unfortunately, the PHP SAX parser does not collapse all contiguous character nodes in a single one, so the character content must be accumulated in the `$chars` variable which is then output before dealing with a start or end tag.

Example 10.18. [CompactHandler.php] PHP SAX handler for text compacting an XML file such as that of Example 2.2

Compare this program with Example 8.4.

```
1 <?php
  class CompactHandler {
    var $closed, $indent, $chars; ❶

5    function CompactHandler(){ ❷
        $this->closed=false;
        $this->indent=0;
        $this->chars="";
    }

10    function startElement($parser, $localname, array $attributes){ ❸
        $this->flushChars();
        if ($this->closed) {
            print "\n".str_repeat(' ', $this->indent);
15            $this->closed=false;
        }
        $this->indent+=1+strlen($localname); ❹
        print $localname."[";
        $first=true;
20        foreach ($attributes as $attrName => $attrValue){ ❺
            if(!$first)print "\n".str_repeat(' ', $this->indent);
            print "@".$attrName."[".$attrValue."]";
            $first=false;
            $this->closed=true;
25        }
    }

    function endElement($parser, $localname){ ❻
        $this->flushChars();
30        print "]";
        $this->closed=true;
        $this->indent-=1+strlen($localname);
    }

35    function characters($parser, $text){ ❼
        if(strlen(trim($text))>0){
            if ($this->closed) {
                print "\n".str_repeat(' ', $this->indent);
                $this->closed=false;
40            }
            $this->chars=$this->chars.trim($text);
        }
    }

45    function flushChars(){ ❽
        if (strlen($this->chars)>0) {
            if ($this->closed) {
                print "\n".str_repeat(' ', $this->indent);
                $this->closed=false;
50            }
            print preg_replace("/ *\n? +/", " ", $this->chars);
            $this->closed=true;
            $this->chars="";
        }
    }
}
```



```

55     }
    }
?>

```

- ❶ Declares variables to save the state of the parsing processing between calls to parsing call-back methods.
- ❷ Initializes two instance variables needed to output the element with the correct indentation and one to accumulate the character content of successive character nodes.
- ❸ When a new element is started, outputs the content of last character nodes and finishes the current indentation if needed.
- ❹ Updates the current indentation by adding the length of the element name.
- ❺ Prints the first attribute on the same line and the others on the subsequent lines properly indented.
- ❻ When an element finishes, outputs the contents of the last character nodes and closes the current bracket and updates the current indentation.
- ❼ For a non-empty text node, ends current line if needed and adds its contents to the `$chars` string.
- ❽ Ends the current indentation and output the normalized content of the `$chars` variables and resets it to an empty string.

10.3.3. StAX parsing using PHP

Compare this program with Example 8.5.

Pull parsing in PHP is done using an instance of the `XMLReader` class which defines a method to move the cursor forward in the XML file (`read()`) and attributes to test the content of the current token (`nodeType`, `hasAttributes` and `name`). Contrarily to the standards, the PHP pull parser does not return a start and end tag for an empty element, but it instead only a single element that can be checked with the `isEmptyElement` method.

Example 10.19. [StAXCompact.php] Text compaction of the cellar book (Example 2.2) with PHP using the StAX model

```

1 <?php
  function compact_($xmlsr,$indent){
    if($xmlsr->nodeType==XMLReader::ELEMENT){
      print $xmlsr->name."[";
5     $indent.=str_repeat(' ',strlen($xmlsr->name)+1);
      $first=true;
      if($xmlsr->hasAttributes){
        while($xmlsr->moveToNextAttribute()){
          if(!$first)print "\n$indent";
10         print "@".$xmlsr->name."[".$xmlsr->value."]";
          $first=false;
        }
      } // Warning: PHP does not return an END_ELEMENT on a empty tag
      if(!$xmlsr->isEmptyElement)
15         while(true){
          do {$xmlsr->read();}
          while ($xmlsr->nodeType==XMLReader::SIGNIFICANT_WHITESPACE);
          if($xmlsr->nodeType==XMLReader::END_ELEMENT)break;
          if($first)$first=false;
20         else print "\n$indent";
          compact_($xmlsr,$indent);

```

```

        }
        print "]";
    } else if ($xmlsr->nodeType==XMLReader::TEXT)
25     print preg_replace("/ *\\n? +/", " ", trim($xmlsr->value));    ❹
    else {
        die("STRANGE NODE:". $xmlsr->nodeType. "\\n");
    }
}
30
$xmlsr = new XMLReader();    ❺
$xmlsr->open("php://stdin");
$xmlsr->setParserProperty(XMLReader::SUBST_ENTITIES,true);
while ($xmlsr->nodeType!=XMLReader::ELEMENT)$xmlsr->read();    ❻
35 compact_($xmlsr, "");    ❼
print "\\n";
?>

```

- ❶ Method to compact from the current token. As there is already a `compact` method predefined in PHP, we changed the name of our method by adding a trailing underscore.
- ❷ If it is a start element tag, outputs the name of the element followed by an opening bracket and update the current indentation.
- ❸ Outputs each attribute name and value all indented except the first one. Attributes are obtained by iteration using the `moveToNextAttribute` method.
- ❹ Loops on children nodes that are not whitespace and compacts each of them with the correct indentation. Some care must be taken not to call this process on an empty tag (i.e. without any children node) because the PHP parser does not return the corresponding end tag.
- ❺ Recursive call to the compacting process.
- ❻ Prints the normalized character content.
- ❼ Creates a new stream parser, indicates that it will parse the standard input and that entities must be substituted by the parse.
- ❽ Ignores the tokens that come before the first element (e.g. processing instructions).
- ❾ Calls the compacting process with the current token and then prints a newline to flush the content of the last line.

10.3.4. Creating an XML document using PHP

In order to demonstrate the creation of new XML document, we will parse the compact form produced in Section 10.3.1 or in Section 10.3.2 like we did in Chapter 9. We first need a way to access appropriate *tokens* corresponding to important signals in the input file. This is achieved by defining a `CompactTokenizer` class (Example 10.20) that will return opening and closing square brackets, at-signs and the rest as a single string. Newlines will also delimit tokens but will be ignored in the document processing. The file is processed by a series of calls to `nextToken` or `skip`. `skip` provides a useful redundancy check for tokens that are encountered in the input but are ignored for output.

Example 10.20. [`CompactTokenizer.php`] Specialized string scanner that returns tokens of compact form.

Compare this program with Example 9.1.

```

1 <?php
  class CompactTokenizer {
      var $index, $tokens;

5     function CompactTokenizer($file){
        global $index,$tokens;
        $tokens = preg_split("/\n|(\[|\]|@)/",
            file_get_contents($file),
            -1,PREG_SPLIT_NO_EMPTY|PREG_SPLIT_DELIM_CAPTURE);
10     $index = 0;
    }

    function getToken(){
        global $tokens,$index;
15     $token = $index<count($tokens)?$tokens[$index]:false;
        return $token;
    }

    function nextToken(){
20     global $tokens,$index;
        do {
            $index=$index+1;
            $token = $this->getToken();
            if($token==false)break;
25     $token=trim($token);
        } while (strlen($token)==0);
        return $token;
    }

30     function skip($sym){
        if($this->getToken()==$sym)
            return $this->nextToken();
        die("skip:$sym expected but ".$this->getToken()." found. index=$index");
    }
35 }
?>

```

- ❶ Defines the CompactTokenizer class with two instance variables \$tokens an array to keep all tokens and \$index to indicate the current token.
- ❷ Initializes the \$tokens array in which the string of the whole file is split according to a regular expression; the last parameter sets flags so that no empty string tokens are of the preg_split method is a combination of flags to indicate that the delimiters are also returned as tokens and that no empty string are returned as tokens. The index of the current token is set to the first element of the array
- ❸ Check that the index of the current token is within the bounds of the array and set it to false otherwise. Return it.
- ❹ Gets the next token but skip those containing only whitespace. It returns the value of the current token.
- ❺ Checks that the current token is the same as the one given as parameter and retrieves the next one. If the current token does not match, then stop the program by indicating the expected token and the one found with the content of the current line.

In PHP, the node creation and child addition is using DOM methods `createElement()`, `appendChild()` and `createTextNode()`. The attributes of a node are added using the `setAttribute()` method.

Example 10.21. [DOMExpand.php] PHP compact form parsing to create an XML document

A sample input for this program is Example 5.8 , which should yield Example 2.2. Compare this program with Example 9.2.

```

1  <?php
    require 'CompactTokenizer.php'; ❶

    function expand($elem){ ❷
5     global $st,$dom;
        $st->skip('[');
        while($st->getToken()=='@'){ // process attributes ❸
            $attName=$st->skip("@");
            $st->nextToken();
10         $elem->setAttribute("$attName",$st->skip("["); ❹
            $st->nextToken();
            $st->skip("]");
        }
        while ($st->getToken()!==false && $st->getToken()!="]") { ❺
15         $s = trim($st->getToken()); // process children
            $st->nextToken();
            if($st->getToken()=="[" ❻
                expand($elem->appendChild($dom->createElement($s))); ❻
            else ❼
20         $elem->appendChild($dom->createTextNode($s)); ❼
            }
            $st->skip("]");
        }
    }

25 $st = new CompactTokenizer('php://stdin'); ❽
    $dom = new DOMDocument(); ❾
    $rootname = $st->getToken(); ❿
    while($st->nextToken()!='[')
        $rootname=$st->getToken();

30 expand($dom->appendChild($dom->createElement($rootname))); ⓫

    $dom->formatOutput=true; ⓫
    print $dom->saveXML();

35 ?>
```

- ❶ Ensures that the `CompactTokenizer` (Example 10.20) is loaded.
- ❷ Adds the content of the file corresponding to the children of the current node to the `$elem` element.
- ❸ Because all attribute names start with an `@`, loops while the current token is equal to `@`. The name of the attribute is saved and the following opening square bracket is skipped, the value is kept and the `]` is skipped.
- ❹ Adding an attribute is done by calling the `setAttribute` method of the current element.

- ⑤ All children are then processed in turn until a closing square bracket is encountered. `s` is the current token, which is either a child element name (if followed by an opening square bracket) or the content of a text node.
- ⑥ A new element named `s` is created and its children are filled in by a recursive call to `expand`.
- ⑦ A text node is added as the last child of the current element.
- ⑧ Creates the tokenizer on the standard input.
- ⑨ Initializes a new XML document.
- ⑩ The name of the root is the first token immediately followed by an opening square bracket.
- ⑪ Creates the root node which is filled in by a call to `expand`.
- ⑫ Serializes the document on the standard output. Sets a flag so that the XML is properly indented. Serialization is done using the `saveXML` method of the document node.

10.3.5. Other means of dealing with XML documents using PHP

PHP also allows other ways of dealing with XML files, the simplest of which is to apply an XSLT stylesheet on a loaded DOM structure. Currently, only XSLT 1.0 stylesheets can be used without having to link with external Java classes using a *PHP/Java bridge*. The following example shows how to perform the transformation defined by the Example 5.9 stylesheet. As the PHP transformation is performed by `xsltproc` that does not deal very well with entities, for this case, we had to modify our original stylesheet Example 5.9 by replacing all entities references by their values. The resulting XSLT `compact-php.xsl` is not given here but it is available on the companion web site.

Example 10.22. [`xslcompact.php`] PHP compaction of an XML file using an XSLT stylesheet.

Application of an XSLT stylesheet on an XML file. Here we use a modified version Example 5.9 to produce a compact version of an XML file.

```

1 <?php
  $xml = new DOMDocument();
  $xml->load("php://stdin");
  ①

5 $xsl = new DOMDocument();
  $xsl->load('compact-php.xsl');
  ②

  $proc = new XSLTProcessor();
  $proc->importStyleSheet($xsl);
  ③
10 $proc -> setParameter(null, 'name', 'value'); // unused in this example

  print $proc->transformToXML($xml);
  ④
?>
```

- ① Initializes the DOM structure for the instance file to transform and loads the file, here we deal with the standard input.
- ② Initializes the DOM structure for the transformation stylesheet and loads it.
- ③ Configures the transformation processor with the stylesheet and sets some parameters (unused here)
- ④ Serializes the output of the transformer on the instance file.

For simple reading and manipulation of information of an XML file, one might consider the SimpleXML extension which, upon loading an XML file, translates its the tree structure into a PHP object, an instance of the SimpleXMLElement class, that can be processed using the usual property selectors and array iterators. As in PHP arrays are zero-based contrarily to XPath in which elements are numbered starting at 1, some care must be given when translating the expressions from one to the other. It is also possible to use XPath expressions to select elements in this structure. Methods are provided to get the name of an element, its attributes and its children nodes. Although it is possible to deal with namespaces, their use is error-prone. The type of object created by SimpleXML is particular, so its uses should be kept for simple operations and extraction of XML data. When an SimpleXMLElement instance is compared or combined with other variables such as integers or strings, it must be casted into the appropriate type before being used. Moreover, SimpleXML has no provision for mixed content, so for generality the DOM approach should be preferred, but it can be useful in some cases.

Example 10.23 shows the use of SimpleXMLExpressions to access some information in Example 2.2. The expressions correspond to the examples given in Example 4.1.

Example 10.23. [simpleXMLPath.php] SimpleXML file loading followed by PHP SimpleXML expressions

SimpleXML file loading followed by PHP expressions corresponding to the XPath expressions of Example 4.1.

```
1 <?php
  $cellarbook = simplexml_load_file("CellarBook.xml"); ❶

  p("/cellar-book/owner", ❷
5   $cellarbook->owner);
  p("/cellar-book/cellar/wine[quantity<2]",
    $cellarbook->xpath("/cellar-book/cellar/wine[quantity<2]"));
  p("/cellar-book/cellar/wine[1]",
    $cellarbook->cellar->wine[0]);
10 p("//postal-code/..",
    $cellarbook->xpath("//postal-code/.."));
  p("/cellar-book/owner/street",
    $cellarbook->owner->street);
  foreach ($cellarbook->cellar->wine as $w) {
15   p("//wine/@code", $w["code"]);
  }
  foreach ($cellarbook->children("http://www.iro.umontreal.ca/lapalme/wine-catalog")->
    children() as $w){p("//cat:wine/@code", $w["code"]);
  }
20 p("/cellar-book/cellar/wine[1]/comment",
    $cellarbook->cellar->wine[0]->comment);
  $sum=0;
  foreach($cellarbook->cellar->wine as $w){
    $sum+=(int)$w->quantity;
25 }
  p("sum(/cellar-book/cellar/wine/quantity)", $sum);

  print("for \$w in //wine return
    concat(\$w/quantity, ':', //cat:wine/@code[.=\$w/@code]/../@name)
```

```

30 ");
    foreach($cellarbook->cellar->wine as $w){
        foreach ($cellarbook->
            children("http://www.iro.umontreal.ca/lapalme/wine-catalog")
                ->children() as $catw){
35         if((string)$w["code"] == (string)$catw["code"]){
            print($w->quantity." : ".$catw["name"]."\n");
        }
    }
    print("-----\n");
40
    $cheapFrenchWines=array();
    foreach ($cellarbook->children("http://www.iro.umontreal.ca/lapalme/wine-catalog")
        ->children() as $catw){
        if((string)$catw->origin->country=="France" && (int)$catw->price<20)
45         array_push($cheapFrenchWines,$catw["name"]);
    }
    p("//cat:wine[cat:origin/cat:country='France' and cat:price&lt;20]",
        $cheapFrenchWines);

50 function p($xpath,$sxo){
    print("$xpath\n");
    var_dump($sxo);
    print("-----\n");
}
55
?>

```

- ❶ Loads the file given as parameter and returns the root element.
- ❷ List of example expressions. As comparison, the corresponding XPath is given as a string, followed by the SimpleXML expression which it printed within the p method defined below.
- ❸ Prints the string with the XPath expression, the structure returned by the evaluation of the expression. var_dump is used because it indicates the precise type of the expression. A separator line is printed to delimit each example.

In order to illustrate the manipulation of SimpleXML structures, we give in Example 10.24 a version to produce a compact version of an XML file. We simply parse (load) the file and the SimpleXML structure is built in memory. It is then a simple matter of traversing this structure to produce a compact version of the file. As a SimpleXMLElement instance has only one slot for its character content, it cannot cope correctly with mixed content elements. Some information is lost in this process. For example, all the text nodes in the food-pairing element of the first wine in the catalog will be concatenated as a single text node. When the SimpleXML is printed in compact form, then first the text child is printed and then the element children afterwards.

Example 10.24. [SimpleXMLCompact.php] PHP compaction of an XML file using a SimpleXML.

Compaction of an XML file by first creating a SimpleXML object and the traversing it with standard PHP iterators to create a compact version of the file.

```

1 <?php
  function compact_($sxelem,$indent){
    print $sxelem->getName()."[";
    $indent.=str_repeat(' ',strlen($sxelem->getName()+1));
5   $first=true;
    foreach ($sxelem->attributes() as $attname => $attvalue) {
      if(!$first)print "\n$indent";
      print "@{$attname}[$attvalue]";
      $first=false;
10  }
    $text = trim($sxelem);
    if(strlen($text)){
      print preg_replace("/ *\n? +/", " ", $text);
      $first=false;
15  }
    $children = $sxelem->children();
    foreach ($children as $child){
      if(!$first)print "\n$indent";
      compact_($child,$indent);
20  $first=false;
    }
    print "]"";
  }

25 $dom = simplexml_load_file('php://stdin');
   compact_($dom,"");
   print "\n";
   ?>

```

- ❶ Function to compact a simple xml object (first parameter) with each new line preceded by an indentation given by the second parameter. As `compact` is a predefined function in PHP for dealing with arrays, we rename the function to `compact_`.
- ❷ Prints the name of the element followed by an open bracket and updates the indentation by adding spaces corresponding to the number of characters in the element name.
- ❸ Prints the array of attributes on different lines except for the first. The name of the attribute is preceded by `@` and the value is put within square brackets.
- ❹ If there is text content, prints it. The text is normalized by replacing contiguous spaces and new lines by a single space. Spaces at the start and end are removed first. Beware that the content is the concatenation of all child text nodes, so this is problematic in the case of mixed content elements.
- ❺ if the element has children, compacts them, a new line is output if it is not the first child.
- ❻ Calls `compact_` recursively.
- ❼ Reads an XML instance file, here the standard input and keeps a reference on the root node.
- ❽ Calls the compacting function on the root element with an empty indentation.

Example 10.25 follows the same structure as Example 10.21 to expand a compact form into a `SimpleXMLElement` structure. It uses the tokenizer of Example 10.20 to read the file. It recursively creates the XML structure as it processes the file. As the PHP library does not provide an indented form of the XML string, we provide one here as another illustration of `SimpleXMLElement` use.

Example 10.25. [SimpleXMLElement.php] PHP compact form parsing to create a SimpleXMLElement document

A sample input for this program is Example 5.8 , which should yield a file equivalent to Example 2.2 except for the mixed content elements. Compare this program with Example 9.2.

```

1  <?php
    require 'CompactTokenizer.php'; ❶

    function expand($elem){ ❷
5     global $st;
        $st->skip([' ']);
        while($st->getToken()=='@'){ // process attributes ❸
            $attName=$st->skip("@");
            $st->nextToken();
10         $elem->addAttribute("$attName",$st->skip("[ "]); ❹
            $st->nextToken();
            $st->skip("]");
        }
        while ($st->getToken()!=false && $st->getToken()!="]") { ❺
15         $s = trim($st->getToken()); // process children
            $st->nextToken();
            if($st->getToken()=="[") {
                expand($elem->addChild($s)); ❻
            } else
20             $elem[0]=$s; // add a text node ❼
        }
        $st->skip("]");
    }

25 $st = new CompactTokenizer('php://stdin'); ❽
    $rootname = $st->getToken(); ❾
    while($st->nextToken()!='['){
        $rootname=$st->getToken();
    }
30
    $sxelem = new SimpleXMLElement("<$rootname/>"); ❿
    expand($sxelem); ⓫

    print ppXML($sxelem,""); ⓬
35
    function ppXML($sxelem,$indent){ ⓭
        $res="";
        $name = $sxelem->getName();
        $res."$indent<$name";
40         if(count($sxelem->attributes())>0) ⓮
            foreach ($sxelem->attributes() as $attname => $attvalue)
                $res.=" ".$attname.'"'.$attvalue.'"';
        $text = trim((string) $sxelem);
        if($sxelem->count()==0 && strlen($text) == 0)
45         $res."/>\n"; ⓯
        else if($sxelem->count()==0) ⓰
            $res.">$text</$name>\n";
    }

```

```
        else {
            $res.=">$text\n";
50         foreach ($sxelem->children() as $schild)
            $res.=ppXML($schild,$indent."  ");
            $res.=" $indent</$name>\n";
        }
        return $res;
55 }

?>
```

- ❶ Ensures that the `CompactTokenizer` (Example 10.20) is loaded.
- ❷ Adds the content of the file corresponding to the children of the current node to the `$elem` element.
- ❸ Because all attribute names start with an `@`, we loop while the current token is equal to `@`. The name of the attribute is saved and the following opening square bracket is skipped, the value is kept and the `]` is skipped.
- ❹ Adding an attribute is done by calling the `setAttribute` method of the current element.
- ❺ All children are then processed in turn until a closing square bracket is encountered. `s` is the current token, which is either a child element name (if followed by an opening square bracket) or the content of a text node.
- ❻ A new element named `s` is created and its children are filled in by a recursive call to `expand`.
- ❼ A text node is added as the child of the current element. As there is no explicit method to specify text content, we assign the text content to the first array position of the `SimpleXMLElement`. This has the unfortunate side-effect of removing the previous child nodes in the case of mixed content.
- ❽ Creates the tokenizer on the standard input.
- ❾ The name of the root is the first token immediately followed by an opening square bracket.
- ❿ Creates the root node as a `SimpleXMLElement`.
- ⓫ The root node which is filled in by a call to `expand`.
- ⓬ Serializes the document on the standard output by calling the `ppXML` method defined below. It is similar to the `asXML` method of the `SimpleXMLElement` but it returns an *indented* string.
- ⓭ Recursive function to create and indented string for the current element. The `$indent` parameter is a string which is output before the start of each new line. The output string is built in the `$res` variable.
- ⓮ Outputs the name of element followed by its attributes on a single line.
- ⓯ If there is no text content and no children nodes, closes the current tag.
- ⓰ If there are not children node, outputs the text and closes the tag.
- ⓱ Close the current tag followed by possible text. Recursively process each children and concatenates their result in the current string output.

10.4. XML processing with JavaScript

XML processing with JavaScript in the context of a browser is somewhat simplified as it can leverage the DOM API already part of the browser for dealing with HTML which is a dialect of XML. This is the approach we present in this section. To use JavaScript outside of a web browsing context, then one should consider using the `jsdom` [70] package for Node.js.

Parsing a string containing an XML structure to get the corresponding DOM structure is done by calling

```
new DOMParser().parseFromString(xmlStr, "text/xml")
```

which returns a DOM structure that can be manipulated using similar calls as in Java. For example, `children` to get the list of children elements, `attributes` to get a mapping of attribute names with the corresponding values, `appendChild(node)` to add a new child to the current element, etc.

Once the DOM structure has been modified, a string with the XML structure can be obtained by calling

```
new XMLSerializer().serializeToString(document)
```

The resulting string is a correct XML structure, but is not indented so some further processing is needed to get a more readable display. One way is to use an XSLT transformation within the browser, but browsers often differ on how they handle such transformations. See this example for such an approach. We instead use a specialized string formatter.

We illustrate JavaScript XML processing through a very simple web page (see Figure 10.1) that embeds our running example, compacting and expanding XML structures.

Figure 10.1. Display of the webpage for exercising JavaScript compaction and expansion.

Compaction and expansion of XML

XML	Compact form
<pre><wine-list> <wine name="Domaine de l'île Margaux" appellation="Bordeaux supérieur"> <is-red>true</is-red> <origin> <country>France</country> <region>Bordeaux</region> </origin> <price>22.80</price> <year>2002</year> </wine> <wine name="Riesling Hugel" appellation="Alsace"> <is-red>>false</is-red> <origin> <country>France</country> <region>Alsace and East</region> </origin> <price>17.95</price> <year>2002</year> </wine> </wine-list></pre>	<pre>wine-list[wine[@name[Domaine de l'île Margaux] @appellation[Bordeaux supérieur] is-red[true] origin[country[France] region[Bordeaux]] price[22.80] year[2002]] wine[@name[Riesling Hugel] @appellation[Alsace] is-red[false] origin[country[France] region[Alsace and East]] price[17.95] year[2002]]]</pre>
<p><input type="button" value="compact"/> <input type="button" value="pretty-print"/></p>	<p><input type="button" value="expand"/></p>

The left part shows the XML content and the right part shows the compact form once the user has clicked on the compact button.

Figure 10.2. HTML to create the web page for compaction and expansion

```

1  <!DOCTYPE html>
   <html lang="en">
   <head>
       <title>Compaction and expansion of XML</title>
5   <script src="https://code.jquery.com/jquery-latest.min.js"></script>
       <script src="DOMCompact.js"></script>
       <script src="CompactTokenizer.js"></script>
       <script src="DOMExpand.js"></script>
       <script src="FormatXML.js"></script>
10  <style>
       textarea{font-family: 'Courier New', Courier, monospace;font-size: large;}
       .center {text-align: center;}
   </style>
</head>
15 <body>
       <h1>Compaction and expansion of XML</h1>
       <table>
           <tr><th>XML</th><th>Compact form</th></tr>
           <tr>
20          <td><textarea name="xml" id="xml" cols="50" rows="20"></textarea></td>
              <td><textarea name="text" id="text" cols="50" rows="20"></textarea></td>
           </tr>
           <tr>
25          <td class="center">
              <input type="button" id="compact" value="compact">
              <input type="button" id="pprint" value="pretty-print">
           </td>
              <td class="center"><input type="button" id="expand" value="expand"></td>
           </tr>
30  </table>
</body>
</html>

```

- ❶ HEAD section that defines the title of the page, makes links to JavaScript files and add some elementary formatting using CSS.
- ❷ body with a title and table containing two text boxes and buttons.
- ❸ Text box in which the XML content can be pasted.
- ❹ Text box where the compaction will appear.
- ❺ Buttons for launching the compaction, expansion and pretty-print of the expanded XML.

10.4.1. DOM parsing using JavaScript

Example 10.26. [DOMCompact.js] Text compaction of the cellar book (Example 2.2) with JavaScript using the DOM model

Compare this program with Example 8.1.

```

1 // taken from https://developer.mozilla.org/en-US/docs/Web/API/Node/nodeType
  const ELEMENT_NODE = 1;
  const TEXT_NODE = 3;
  const COMMENT_NODE = 8;
5
  function domCompact(node, indent){
    const nodeType = node.nodeType;
    switch (nodeType) {
      case ELEMENT_NODE:
10      let out=[]; // create list of compacted strings
        indent+=" ".repeat(node.nodeName.length+1)
        const attributes = node.attributes;
        for (let i = 0; i < attributes.length; i++) {
15          const attr = attributes[i];
          out.push(`@${attr.nodeName}[${attr.nodeValue}]`);
        }
        const children=node.childNodes;
        for (let i = 0; i < children.length; i++) {
20          const child = children[i];
          // skip comment and empty text nodes
          if (child.nodeType===COMMENT_NODE ||
              (child.nodeType===TEXT_NODE &&
               child.textContent.trim().length==0))continue;
          out.push(domCompact(child,indent));
25        }
        return `${node.nodeName}[${out.join("\n"+indent)}]`;
      case TEXT_NODE:
        return node.textContent.trim();
      default:
30        console.log("Should never happen: bad nodeType",nodeType)
    }
  }
  $(document).ready(function(){
    $("#compact").click(function(){
35      const xmlStr=$("#xml").val();
      const doc=new DOMParser().parseFromString(xmlStr,"text/xml");
      $("#text").val(domCompact(doc.documentElement,""));
    })
  })

```

- ❶ Constants for relevant DOM node types.
- ❷ Start of function to produce a compact string from a node with a given indent (a string of spaces).
- ❸ Determines the node type and switch to the appropriate code. Only element and text types are dealt with.
- ❹ If the node is an element, a list of compact string is initialized for adding attributes and children nodes. The new indentation value is computed for the children nodes.
- ❺ Each attribute is added with its name preceded by @ and its value within square brackets.
- ❻ Each child node is added with a recursive call to `compact`.
- ❼ Returns a new string in which the compacted strings for the attributes and children are joined with a new line and the appropriate indentation.
- ❽ A text node is the *trimmed* content of the node.

- ⑨ The callback function for the `compact` button gets the XML string in the left text box, parses it to produce an XML document which is then compacted and added to textbox on the right.

10.4.2. Creating an XML document using JavaScript

Parsing a compact form in order to create an XML document follows an organization similar to the one we have shown in Chapter 9 and Section 10.1.3. We first define an object for a specialized tokenizer which returns only meaningful units that will be used to create the XML elements and attributes of the resulting XML document.

Example 10.27. [CompactTokenizer.js] JavaScript specialized string scanner that returns tokens of compact form

Compare this program with Example 9.1.

```
1 function CompactTokenizer(source) {                                ❶
    this.token="";                                                ❷
    this.tokens=source.split(/([\|\|\\\n|@[^\[\]\n@]+)/) // separate all tokens
    .filter((e)=>e.length>0 && !e.match(/^\\n? *$/)) // keep only non empty tokens
5    );

    this.nextToken = function() {                                    ❸
        return this.token = this.tokens.length==0 ? null : this.tokens.shift();
    }
10
    this.skip=function(sym) {                                       ❹
        if(this.token==sym)
            return this.nextToken();
        else
15         throw new ArgumentError("skip:"+sym+" expected but "+
            this.token+ " found");
    }
}
```

- ❶ Constructor for building a specialized tokenizer for the string given as source.
- ❷ Keeps the current token.
- ❸ Builds the array of all tokens by *splitting* according to important elements of the inputs. Here we are interested in the content matched by the regular expressions, so we *capture* the content matched. However as the input is matched by the regular expression, empty strings between each match are produced. These empty matches and those containing only spaces are removed by the call to `filter`.
- ❹ Retrieves the first element of the `tokens` array, removes it and saves it as the current token, which is then returned as the value of the call.
- ❺ Checks that the current token corresponds to the parameter. If this is the case, returns the next token, otherwise raises an exception. In principle, this exception should never be raised but this function might be useful (and it was!) for debugging purposes.

The creation of a new XML document is done by a call to the `domExpand` function which launches a series of recursive calls to `expand`.

Example 10.28. [DOMExpand.js] JavaScript compact form parsing in order to create an XML document

A sample input for this program is Example 5.8 to yield Example 2.2. Compare this program with Example 9.2.

```

1 function expand(st,doc,defaultNS,elementName){                                ❶
    let elem;
    const colonPos=elementName.indexOf(":")                                  ❷
    if (colonPos>0) // qualified element name
5      elem = doc.createElementNS(elementName.substring(0,colonPos),
                                     elementName.substring(colonPos+1))
    else
        elem=doc.createElementNS(defaultNS,elementName);
    st.skip("[");
10  while (st.token=="@"){ // process attributes                               ❸
        const attName=st.skip("@");
        st.nextToken();
        const attValue=st.skip("[");
        if (attName=="xmlns")defaultNS=attValue; // set new default namespace
15  elem.setAttribute(attName,attValue);
        st.nextToken();
        st.skip("]")
    }
    while (st.token!="]"){                                                    ❹
20      const str=st.token.trim();
        st.nextToken();
        if (st.token=="["){
            elem.appendChild(expand(st,doc,defaultNS,str))                    ❺
        } else {
25      elem.appendChild(doc.createTextNode(str))
        }
    }
    st.skip("]");
    return elem;
30 }

function domExpand(s){                                                       ❻
    let st=new CompactTokenizer(s);
    let rootName;
35  while(st.nextToken()!="[")rootName=st.token;
    let doc= new Document()
    doc=expand(st,doc,"",rootName);
    return doc;
}
40

$(document).ready(function(){                                               ❼
    $("#expand").click(function(){                                           ❽
        const text=$("#text").val();
        const xmlStr=new XMLSerializer().serializeToString(domExpand(text))
45  $("#xml").val(xmlStr)
    })
    $("#pprint").click(function(){                                           ❾

```

```
        const xmlStr=$( "#xml" ).val();  
        $( "#xml" ).val( formatXml( xmlStr ) )  
50    } )  
    } )
```

- ❶ Defines the function with the following parameters: the `CompactTokenizer` object, the document node necessary for creating new elements, the default namespace and the element name to be created during expansion.
- ❷ Creates a new element checking if the element name is *qualified*, in which case it sets the appropriate namespace on creation otherwise it uses the default namespace.
- ❸ Each attribute is created with a call to `setAttribute`. The element name is the result of the call to `skip("[")` which returns the token after the opening bracket. If the name of the attribute is `xmlns`, it resets the default namespace.
- ❹ Loops on each child. Gets the string that follows, which is then *trimmed*.
- ❺ If the current token is an opening bracket, then we create a new element by a recursive call to `expand` which is added as a child of the current node. Otherwise the string is added as a text node.
- ❻ The function receives a string with the compact form for which the `CompactTokenizer` is created. The root element name is the identifier preceding the first left bracket. A new document object is then created and given as parameter to the `expand` function.
- ❼ Associates callback functions with the `expand` and `pretty-print` buttons.
- ❽ Function that gets the value of the expand string (the right text box), gives it to the `domExpand` function whose result is serialized in the left box. Unfortunately, this result is not indented.
- ❾ Function that gets the value of the raw XML and formats it using a function that it not shown here.

10.5. XML processing with Swift

Swift [85] is a statically typed script programming language with a syntax inspired by functional languages such as Haskell and script like Python. In this section, we will show how to use it for both DOM and SAX parsing, which we used in XML processing in Java.

10.5.1. DOM parsing using Swift

To show how to process an existing XML structure, we will use the compacting process that we programmed in Java in Section 8.1. In Swift, DOM parsing is achieved simply by calling the `XMLDocument` constructor available as `Foundation` class. This creates a DOM structure composed of instances of the `XMLNode` class.

Example 10.29. [DOMCompact.swift] Text compaction of the cellar book (Example 2.2) with Swift using the DOM model

Compare this program with Example 8.1.

```

1 import Foundation ❶

   func processDOM(_ string:String)->String{ ❷
       var res=""

5
       func compact(_ nodeIn:XMLNode?,_ indent:String){ ❸
           guard (nodeIn != nil) else {return}
           let node = nodeIn!
           var newIndent=indent
10          switch node.kind {
               case .element: ❹
                   let name = node.name!
                   let elem = node as! XMLElement
                   res += name+"["
15                   newIndent += String(repeating:" ", count: name.count+1)
                   var first = true
                   if elem.attributes != nil { ❺
                       for attr in elem.attributes! {
                           if !first {res += "\n\(newIndent)"}
20                           res += "@\(attr.name!)[\(attr.stringValue!)]"
                           first = false
                       }
                   }
                   if elem.children != nil { ❻
25                       for child in elem.children! {
                           if !first {res += "\n\(newIndent)"}
                           compact(child, newIndent) ❼
                           first = false
                       }
30                   }
                   res += "]"
               case .text: ❽

```

```

        res += node.stringValue!.trimmingCharacters(in:
                                                    .whitespacesAndNewlines)
35     default:
        print("compact:unprocessed kind:\(node.kind)")
    }
}

40     do {
        let doc = try XMLDocument(data: string.data(using: .utf16)!, ❹
                                options: [])
        compact(doc.rootElement(), "") ❷
        return res+"\n"
45     } catch {
        print("Error in the input file:\(error)")
        return ""
    }
}

50 func processDOM(path:String)->String { ❶
    return processDOM(try! String(contentsOf: URL(fileURLWithPath: path)))
}

```

- ❶ Imports the Foundation framework needed for XML processing.
- ❷ Function that takes the XML input as a string and returns the compacted form as a string, here the `res` variable.
- ❸ A recursive function that processes an XML node in order to add to `res` indented lines corresponding to the input structure. The second parameter is a string with the current number of spaces for indentation of each new line.
- ❹ If the node is an element, it adds to the the name of the node followed by an opening bracket and adds the appropriate number of spaces to the current indentation.
- ❺ Deals with attributes which are contained in a list of nodes over which we iterate. For each attribute, prints a `@`, its name and the corresponding value within square brackets. A new line is started for each attribute except the first.
- ❻ Loops over the list of children possibly changing line if it not the first child.
- ❼ Recursively calls `compact` on the child node with the updated indentation.
- ❽ Processes a text node by adding it to `res` after removing starting and ending spaces, tabs and newlines.
- ❾ Parses the file by creating a new XML document from the content of the input string.
- ❿ Calls the compacting process on the document node with an empty indentation string and ends the last line with a newline.
- ⓫ Transforms the XML file identified by a path string to produce a string containing the compacted form by calling the `processDOM` function on the string content of the file.

10.5.2. SAX parsing using Swift

In order to process an XML structure with the SAX approach, we will use a similar compacting process to the one we programmed in Java in Section 8.2. SAX parsing is achieved by creating an instance of the `xmlParserDelegate` class in which we define handler methods for some parsing events. The parsing process will send *call-backs* to the handler of these events, called a *delegate* in Swift, to create the compacted document as a string. SAX parsing in Swift is only a matter of defining the appropriate functions in the

delegate. Because nodes are not normalized, many successive text nodes can appear within an element, so some care has to be taken to deal with this fact. Unfortunately, in Swift 5, internal entities such as the ones used at the start of the `CellarBook.xml` file cannot be properly dealt with.

Example 10.30. [SAXCompact.swift] Text compaction of the cellar book (Example 2.2) with Swift using the SAX model

Compare this program with Example 8.3.

```

1 import Foundation ❶

    // Caution: internal entities are not dealt properly with this API.
    func processSAX(string:String)->String {
5
        class xmlParserDelegate:NSObject, XMLParserDelegate { ❷
            var res=""
            var closed=false
            var indent=""
10            var lastNodeWasText=false

            func parser(_ parser: XMLParser, ❸
                didStartElement elementName: String,
                namespaceURI: String?, qualifiedName qName: String?,
                attributes attributeDict: [String : String] = [:]){
15
                if closed {
                    res += "\n\(indent)"
                    closed = false
                }
20                indent += String(repeating: " ", count:elementName.count+1) ❹
                res += "\(elementName)["
                var first = true
                for (name,val) in attributeDict { ❺
                    if !first {res += "\n\(indent)"}
25                    res += "@\(name)[\(val)]"
                    first=false
                    closed=true
                }
                lastNodeWasText=false
30            }

            func parser(_ parser: XMLParser, ❻
                didEndElement elementName: String,
                namespaceURI: String?, qualifiedName qName: String?){
35
                res += "]"
                closed=true
                indent=String(indent.dropLast(elementName.count+1))
                lastNodeWasText=false
            }
40
            func parser(_ parser: XMLParser, ❼
                foundCharacters string: String){
                let s=string.trimmingCharacters(in: .whitespacesAndNewlines)
                if s.count>0 {

```

```

45         if closed && !lastNodeWasText{
            res += "\n\(indent)"
            closed = false
        }
        closed=true
50         res += "\(s)"
    }
    lastNodeWasText=true
}

55     public func parserDidEndDocument(_ parser: XMLParser){           ❸
        res += "\n"
    }

    func parser(_ parser: XMLParser,                                   ❹
60         parseErrorOccurred parseError: Error){
        print("** parseErrorOccurred:\(parseError)")
    }

    func parser(_ parser: XMLParser,
65         validationErrorOccurred validationError: Error){
        print("** validationErrorOccurred:\(validationError)")
    }
}

70     let myDelegate=xmlParserDelegate()                             ❩
    let parser=XMLParser(data: string.data(using: .utf16)!)          ❪
    parser.delegate = myDelegate                                     ❫
    parser.parse()                                                  ❬
    return myDelegate.res
75 }

    func processSAX(path:String)->String{                             ❭
        return processSAX(string:try! String(contentsOf: URL(fileURLWithPath: path)))
    }
80

```

- ❶ Imports the Foundation framework for the XML processing.
- ❷ Defines the constructor of the class and initializes instance variables needed to output the element with the correct indentation and for keeping track of the parsing state. The last flag is used when many successive text nodes appears.
- ❸ When a new element is started, finishes the current indentation if needed.
- ❹ Updates the current indentation by adding the length of the element name.
- ❺ Prints the first attribute on the same line and the others on the subsequent lines properly indented.
- ❻ When an element finishes, closes the current bracket and updates the current indentation.
- ❼ For a non-empty text node, ends current line if needed and if the last node was not a text node. Writes the content of the node and indicates that the last node was a text node.
- ❽ When the document parsing finishes, add a terminating newline.
- ❾ Deals with a parsing error by printing an error message.
- ❿ Creates an instance of the SAX parser delegate.
- ⓫ Creates an instance of the SAX parser.
- ⓬ Sets the parser delegate to which the parsing will send events.

- ⑬ Starts the parsing process and return the string containing the result of the compacting process.
- ⑭ Transforms the XML file identified by a path string to produce a string containing the compacted form by calling the `processSax` function on the string content of the file.

10.5.3. Creating an XML document using Swift

In order to demonstrate the creation of new XML document, we will parse the compact form produced in Section 10.5.1 or in Section 10.5.2 like we did in Chapter 9. We first need a way to access appropriate *tokens* corresponding to important signals in the input file. This is achieved by defining a `CompactTokenizer` class that will return opening and closing square brackets, at-signs and the rest as a single string. Newlines will also delimit tokens but will be ignored in the document processing. The file is processed by a series of calls to `nextToken` or `skip`. `skip` provides a useful redundancy check for tokens that are encountered in the input but are ignored for output.

In Swift, a new document is created using the constructor of the `XMLDocument` class. Adding a new node is done with the `addChild` method that adds the new node as the last child of a given node. The attributes of a node are added using the `addAttribute` method.

Example 10.31. [DOMExpand.swift] Swift compact form parsing to create an XML document

A sample input for this program is Example 5.8 , which should yield Example 2.2. Compare this program with Example 9.2.

```

1 import Foundation ❶

   class CompactTokenizer { ❷
       var tokens:[String]
5       var iTok:Int

       init(string:String){ ❸
           let pattern = "(\\[|\\]|@|^[^\\[\\]|@\\n]+)"
           let regex = try! NSRegularExpression(pattern: pattern, options: [])
10          let nsrange = NSRange(string.startIndex..❹
       let token = tokens[iTok]
       iTok+=1
       return token
   }

25  func skip(_ string:String){ ❺
       if nextToken() != string {

```

```

        print("\((string) expected at position \((iTok-1)")
    }
30 }

    func isAtEnd()->Bool {
        return iTok>=tokens.count
    }
35 }

func expand(string:String) -> XMLNode {
    let ct=CompactTokenizer(string: string)
40
    func expandElem(_ elem:XMLElement,_ iTok:String){
        var tok = iTok
        while tok == "@" {
            let attName=ct.nextToken()
            ct.skip("[")
45
            elem.addAttribute(
                XMLNode.attribute(withName: attName,
                    stringValue: ct.nextToken()) as! XMLNode)
            ct.skip("]")
            tok=ct.nextToken()
50
        }
        while tok != "]" && !ct.isAtEnd() {
            let s=tok
            let nextTok=ct.nextToken()
            if nextTok == "[" {
55
                let child = XMLNode.element(withName: s) as! XMLElement
                expandElem(child,ct.nextToken())
                elem.addChild(child)
                tok = ct.nextToken()
            } else {
60
                elem.addChild(XMLNode.text(withStringValue: s) as! XMLNode)
                tok = nextTok
            }
        }
    }
65
    var rootName=""
    var token=ct.nextToken()
    while token != "[" {
        rootName = token
70
        token = ct.nextToken()
    }
    let root = XMLElement(name: rootName)
    let doc=XMLDocument(rootElement: root)
    expandElem(root,ct.nextToken())
75
    return doc
}

func expand(path:String)->XMLNode {
    return expand(string: try! String(contentsOf: URL(fileURLWithPath: path)))
80 }

```

- ❶ Imports the `Foundation` framework for the XML processing.
- ❷ Defines the `CompactTokenizer` class.
- ❸ Defines the `pattern` regular expression to split an input line on an ampersand, an opening or closing square bracket. The square brackets characters in the regular expression must be preceded by a backslash as square brackets are part of the regular expression language, also used in the same expression. The list of all non-empty tokens is created by matching the regular expression on the whole input string.
- ❹ Returns the next token.
- ❺ Checks that the current token is the same as the one given as parameter and retrieves the next one. If the current token does not match, then print an error message.
- ❻ Checks if there are still tokens left to process.
- ❼ Creates an XML document corresponding to the content of the input string.
- ❽ Creates an instance of the `CompactTokenizer` class that will be used to get the tokens.
- ❾ Adds to an XML element the content starting with the current token `inTok`.
- ❿ Because all attribute names start with an `@`, we loop while the current token is equal to `@`. The name of the attribute is saved and the following opening square bracket is skipped, the value is kept and the `]` is skipped.
- ⓫ Adding an attribute is done by calling the `addAttribute` method having an `XMLNode` as parameter. The attribute is created by calling the `XMLNode.attribute` function with the name of the attribute and its corresponding value.
- ⓬ All children are then processed in turn until a closing square bracket is encountered. `s` is the current token, which is either a child element name (if followed by an opening square bracket) or the content of a text node.
- ⓭ A child node is expanded by a recursive call whose result is added as the last child of the current element.
- ⓮ A text node is added as the last child of the current element.
- ⓯ The name of the root is the first token immediately followed by an opening square bracket.
- ⓰ Create the document node by calling `XMLDocument` with an element that serves as its root.
- ⓱ Calls the `expand` function on the content of the file at the given path. The output of this function is an `XMLNode` that can be easily transformed into a prettyprinted string using the `xmlString(options: .nodePrettyPrint)` method.

10.6. XML processing with E4X

Caution. E4X and ActionScript are now *deprecated* for all practical purposes, so you should consider other tools for processing XML such as the use of JavaScript described in the previous section JavaScript.

As more and more Javascript programs are expected to process XML data, it has been thought useful to provide a simple notation for handling XML data directly within Javascript. ECMAScript for XML (E4X)[66] has been recently standardized to support a natural use of XML data within Javascript programs. Its main innovation is the addition of an XML data type and the possibility of using XML literals that can be accessed directly using the usual *dot notation* used for accessing properties of Javascript objects. It is possible to access the content of an XML node but also to change its value or to add or remove new XML nodes.

For example, it is possible to declare a new wine element directly in a Javascript program using the following notation, which is the same as the usual notation for all XML data:

```
var wine:XML = <wine name="Château La Piquette" format="11">
  <properties>
    <color>red</color>
  </properties>
  <comment>Should be thrown away</comment>
  <year>2007</year>
</wine>;
```

wine is then a Javascript variable whose value can be retrieved or changed using the element name. For example, `wine.properties.color` will return "red". If there are more than one element of the same name, a specific one can be selected by putting its ordinal number (in document order) within square brackets after the name. To change an element or add a new one, it is only a matter of using its name as the target of an assignment. Attributes can be accessed by prefixing their names with an @. The `..` operator is the equivalent of the XPath `//` operator.

Here are a few examples of E4X expressions:

- `wine.comment="I found it excellent"` changes the value of the comment;
- `wine.origin.country="USA"` adds a new element `origin` containing the `country` element;
- `wine.@name` returns the name of the wine ("Château La Piquette");
- `wine..color` returns "red".

There is a limited XPath-like notation to access a list of XML nodes over which it is possible to iterate. For example, for each (`var w in wine.*`) will loop over all attributes of a wine.

```
var elems:XMLList = wine.*;
for(var i=0;i<elems.length();i++){
  ... elems[i]...
}
```

will access all wine elements in turn. `XMLList` is a predefined type defining a list of XML nodes and providing specific methods. In order to simplify programming (in most cases), the E4X specification points out that it *deliberately blurs the distinction between an XMLNode and an XMLList containing a single XMLNode*. This design choice is also made in XSL.

It is also possible to select elements and attributes from a list with a boolean expression evaluated on each element of the list. Figure 10.3 gives a few examples of E4X expressions for accessing information in the cellar book.

Figure 10.3. E4X expression examples applied to Example D.1

These expressions are the same as the ones used in Example 4.1. `cellarBook` is a Javascript variable in which the XML document has been stored. It corresponds to the root node (`/cellar-book`) of the XML document.

```

1 cellarBook.owner                                ❶
  cellarBook.cellar.wine.(quantity<=2)           ❷
  cellarBook.cellar.wine[0]                       ❸
  cellarBook.descendants("*").(elements("postal-code").length())>0) ❹
5 cellarBook.owner.street                         ❺
  cellarBook..wine.@code                          ❻
  var catNS:Namespace                             ❼
    = new Namespace("cat","http://www.iro.umontreal.ca/lapalme/wine-catalog");
  cellarBook.addNamespace(catNS);
10 cellarBook..catNS::wine.@code

  var wines:XMLList = cellarBook..wine;          ❽
  wines[wines.length()-1].@code

15 cellarBook.cellar.wine[0].comment.catNS::bold  ❾

  var sum:Number=0;                               ❿
  for each (var q in cellarBook.cellar.wine.quantity)
    sum += Number(q);
20 sum

  var res:String="";                              ⓫
  for each (var w in cellarBook..wine)
    res+=w.quantity+": "+cellarBook..catNS::wine.(@code==w.@code).@name+"\n";
25 res

  cellarBook..catNS::wine.(catNS::origin.catNS::country=="France"  ⓬
    && catNS::price < 20)

```

- ❶ The owner element of the cellar. Result: node on line 103.
- ❷ The wines for which we have 2 bottles or less. The nodes returned are the wine elements that are filtered with a boolean expression (predicate) in parentheses. The predicate uses `quantity`, an internal element evaluated in the current context of the path specified. Result: nodes on lines 131 and 136.
- ❸ The first wine of the cellar. Result: node on line 120.
- ❹ The elements which contain a `postal-code` element. This is achieved by keeping only descendants for which the list of all `postal-code` elements is not empty. Result: nodes at lines 103 and 113.
- ❺ The street of the cellar's owner. Result: "1234 rue des Châteaux".
- ❻ The value of the `code` attribute for all wines in the cellar. Note the use of `..` to find all descendants of a node, analogous to `//` in XPath. Result: "C00043125", "C00312363", "C10263859", "C00929026".
- ❼ The value of the `code` attribute for all wines in the catalog. Note the use of the namespace prefix that must be defined and then added to the XML object. It is used by prefixing the element name with `::`. Result: "C00043125", "C00042101", "C10263859", "C00312363", "C00929026".
- ❽ The code of the last wine of the cellar, obtained by getting the list of wines and returning the one having an index that is one less than the length of the list. Result: "C00929026".

- ⑨ The `cat:bold` element (note again the use of the namespace prefix) within the comment of the first wine of the cellar. Result: "Guy Lapalme, Montréal" (expanded from the entity `&GL;`).
- ⑩ Total number of bottles in the cellar obtained by iterating over the values of all `quantity` elements of the wines in the cellar. As each XML value is a string, it must first be converted to a number for the sum. Otherwise, `+` would concatenate the strings. Result: 14.
- ⑪ Sequence of 4 strings, each giving the number of bottles of each wine in the cellar, followed by a colon and the name of the corresponding wine. Result: 2:Domaine de l'Île Margaux, 5:Mumm Cordon Rouge, 6:Château Montguéret, 1:Prado Rey Roble.
- ⑫ Sequence of French wines in the catalog costing less than 20 dollars. Result: wines that start on lines 24 and 42.

10.6.1. DOM parsing using E4X

Parsing a string to get an XML value with E4X is achieved by casting the string to XML using the `XML` function. Because the content of a file can be read in a string, we can use this function on the resulting string. Example 10.32 presents an ActionScript file that provides a class with methods for converting a string containing an XML source into a string with the compact notation we have seen in previous chapters. It is meant to be called from a Flash program `XMLProcessing.fla` not described in this document but available on the companion Web site. It that allows the user to select a given file and to display the output of the compacting process in a scrollable window. This application can also transform the XML elements into sprites that can be zoomed on when they are clicked.

Example 10.32. [DOMCompact.as] Text compaction of the cellar book (Example 2.2) with E4X using the DOM model

Compare this program with Example 8.1.

```

1 package{
    public class DOMCompact{                                ❶
        var doc:XML;
2
5        function DOMCompact(s:String){                    ❷
            doc = XML(s);
        }
3
10       public function toString():String {                ❸
            return compact(doc);
        }
4
        var blanks:String = "    ";
        private function spaces(n:int){                    ❹
15         while(blanks.length<n)blanks+="    ";
            return blanks.substr(0,n);
        }
5
20       private function compact(node:XML,indent:String=""):String { ❺
            var type:String = node.nodeType();
            switch (type) {
                case "element":                              ❻
                    var out:String = node.localName()+"[";

```

```

    indent += spaces(node.localName().length+1);
25     var attributes:XMLList = node.*;
    var first:Boolean = true;
    for (var i:int=0;i<attributes.length();i++){
        if(i>0)out+="\n"+indent;
        out+="@"+attributes[i].name()+"["+attributes[i]+"]";
30         first=false;
    }
    for each (var child:XML in node.children()){
        if(!first) out+="\n"+indent;
        out+=compact(child,indent);
35         first=false;
    }
    return out+"]";
    case "text":
        return node[0].toString().replace(/ *\n */g, ' ');
40 }
    return "Should never happen!!!";
}
}
}

```

- ❶ Defines the class with one instance variable, the document of type XML.
- ❷ Constructor of the class that creates an XML document from a string. Parsing is done through the XML function.
- ❸ Produces a *compact* string by calling the `compact` function.
- ❹ Defines as instance variable a string from which the `spaces` function will produce a certain number of spaces with the `substring` function. If `blanks` is not long enough, more spaces are added to the string.
- ❺ Main function for producing a compact version of the XML document, starting with a given node and a certain indent given as a string of spaces. By default, the indent is empty.
- ❻ If the node is an element, we initialize the output string with the name of the element followed with an opening bracket. The length of this string is used as indent to be added to the current one for the subsequent lines.
- ❼ All attributes are output with their names and values within brackets.
- ❽ Each child node is output with a recursive call to `compact`.
- ❾ A text node is the content of the node but *normalized* by replacing newlines and their surrounding whitespaces with a single space.

10.6.2. Creating an XML document using E4X

Parsing a compact form in order to create an XML document follows an organization similar to the one we have shown in Chapter 9 and Section 10.1.3. We first define a class for a specialized tokenizer which returns only meaningful units that will be used to create the XML elements and attributes of the resulting XML document.

Example 10.33. [CompactTokenizer.as] E4X specialized string scanner that returns tokens of compact form

Compare this program with Example 9.1.

```

1 package {
    class CompactTokenizer { ❶
        private var tokens:Array = [];
        private var tok:String = "";
5
        function CompactTokenizer(source:String):void{ ❷
            tokens=source.split(/(\[|\]|\\n|@|[[^\[\]\n@]+)/).filter(notEmpty);
        }
10
        private function notEmpty(element:*,index:int,a:Array):Boolean{ ❸
            return element.length>0 && !element.match(/^\\n? *$/);
        }
15
        public function nextToken():String{ ❹
            return tok = tokens.length==0 ? null : tokens.shift();
        }
20
        public function token():String { ❺
            return tok;
        }
25
        public function skip(sym:String){ ❻
            if(token()==sym)
                return nextToken();
            else
                throw new ArgumentError("skip:"+sym+" expected but "+
                    token()+ " found");
        }
30
    }
}

```

- ❶** Defines a class with two private instance variables: `tokens`, an array of all tokens whose elements will be returned one by one; `tok`, the current token.
- ❷** Builds the array of all tokens by *splitting* according to important elements of the inputs. Here we are interested in the content matched by the regular expressions, so we *capture* the content matched. However as the input is matched by the regular expression, empty strings between each match are produced. These empty matches and those containing only spaces are removed by the call to `filter`.
- ❸** Defines a function to match empty strings and those containing only spaces.
- ❹** Retrieves the first element of the `tokens` array, removes it and saves it as the current token, which is then returned as the value of the call.
- ❺** Returns the current token.
- ❻** Checks that the current token corresponds to the parameter. If this is the case, returns the next token, otherwise raises an exception. In principle, this exception should never be raised but this function might be useful (and it was!) for debugging purposes.

The creation of a new XML document is done by a recursive call to the `expand` function that creates each element of the document in turn.

Example 10.34. [DOMExpand.as] E4X compact form parsing in order to create an XML document

A sample input for this program is Example 5.8 to yield Example 2.2. Compare this program with Example 9.2.

```

1 package{
    public class DOMExpand {                                ❶
        var doc:XML;

5        function DOMExpand(s:String){                    ❷
            var st:CompactTokenizer = new CompactTokenizer(s);
            var rootName:String;
            while (st.nextToken()!="[") rootName=st.token();
            doc = expand(st,rootName);                      ❸
10       }

        public function toString():String {                ❹
            return doc.toXMLString();
15       }

        function expand(st:CompactTokenizer,elementName:String):XML{ ❺
            var elem:XML= <{elementName}/>;
            st.skip("[");
            while(st.token()=="@"){
20                var attName:String=st.skip("@");          ❻
                    st.nextToken();
                    elem.@[attName]=st.skip("[");
                    st.nextToken();
                    st.skip("]");
25            }
            while (st.token()!="]"){                          ❼
                var s:String = st.token().replace(/^\s*(.*?)\s*$/, "$1");
                st.nextToken();
                if(st.token()=="[")
30                    elem.appendChild(expand(st,s));        ❽
                else
                    elem.appendChild(s);                      ❾
            }
            st.skip("]");
            return elem;
35        }
    }
}

```

- ❶ Defines a class with a single instance variable: the XML document to create.
- ❷ The constructor instantiates a compact tokenizer and then finds the first string followed by an opening bracket which will be the name of the root.
- ❸ Creates a new XML structure by calling expand on the name of the root element.
- ❹ The serialization of the XML structure is the string representation of the doc instance variable.

- ⑤ Creates a new element whose name is the second parameter. We use the `st` string tokenizer. We first create an empty element using an XML literal. As the name has to be evaluated, it is put within braces.
- ⑥ Each attribute is created with a single assignment statement making use of the attribute name within brackets. This name is the result of the call to `skip("[")` which returns the token after the opening bracket.
- ⑦ Loops on each child. Gets the string that follows, which is then *trimmed* using a regular expression.
- ⑧ If the current token is an opening bracket, then we create a new element by a recursive call to `expand` which is added as a child of the current node.
- ⑨ If the current token is not an opening bracket, then its value is used to create a text node as a child of the current node.

10.7. XML alternative notations

In this document, we have been promoting XML as the *ideal* exchange language between systems because of its standardization and its wide range of applications. Unfortunately XML needs a complex infrastructure (parser, validator, transformer, etc.) in order to profit from its power. In some cases, this may prove to be an overkill. This is why we will now describe two alternative *human readable and writable* notations for information interchange between systems. This will show that XML is not the only exchange formalism and will highlight the advantages and disadvantages of each approach.

10.7.1. JSON

The AJAX technology, built upon the *XMLHttpRequest* (XHR) system call that exchanges information between a browser and a server in order to update a part of a Web page without reloading it entirely. This allows in certain cases a better user experience (dynamic suggestions, on-the-fly server-side validation, etc.) especially when a Web page is used as interface for an interactive system.

As its name implies, XHR involves an XML information exchange between the server and the browser. XML must then be parsed by the Javascript engine of the browser for each exchange in order to create the corresponding Javascript data structure in memory. If E4X is not implemented in the Javascript interpreter, using a full XML parser may be prohibitive in both time and memory usage for simple data exchange. A simpler alternative has been proposed: *JavaScript Object Notation* (JSON) which is based on the conventional literal object notation of Javascript. In this case, creating a Javascript data structure is just a matter of *evaluating* the received data string from the server. Because the `eval` function is already included in all Javascript interpreters, there is thus no need for an external parser.

JSON encodes the information by means of the primitive data structures of Javascript built on the following types of values:

primitive objects	numbers, strings (between double quotes), booleans (<code>true</code> or <code>false</code>) and <code>null</code> ;
complex objects	arrays (elements between square brackets) or objects (key-value pairs between braces). Arrays or objects elements are separated by a comma. A key is a string and it is separated from the corresponding value by a colon.

Complex objects components are accessed using the *dot notation* as for any other Javascript object. If the name of an element is a simple identifier then it can be used directly (e.g. `wine.origin.country`) but if it contains a dash which is not a legal Javascript identifier, then we must use the indexing mechanism such as `wine["is-red"]`. It is similar to what we have shown for E4X in the previous section, but it does not allow for all XPath-like possibilities provided by the direct XML literal notation.

In order to show the similarities between XML and JSON, we show a small XML wine list in Example 10.35 and its JSON equivalent in Example 10.36. A tree with more than one child is translated into an object with a single key corresponding to the root of the tree and its value is another object containing the attributes and children. The attributes are simple key-value pairs and the children are nodes that are themselves objects. Similarly named nodes are combined within an array to allow indexing. If the content of Example 10.36 is kept in the Javascript variable `s`, then parsing is a simple matter of doing `root = eval(s)`¹. In this

¹As some Javascript parsers will consider the first identifier followed by a colon to be the label of a statement, it is usually safer to use `root = eval("(" + s + ")");` to force the parser to consider the string to be an expression.

case, the name of the second wine of this list can be accessed in Javascript with `root["wine-list"].wine[1].name`.

Example 10.35. [wineList.xml] A small wine list in XML

```

1 <?xml version="1.0" encoding="UTF-8"?>
  <wine-list>
    <wine name="Domaine de l'Île Margaux"
      appellation="Bordeaux supérieur">
5      <is-red>true</is-red>
      <origin>
        <country>France</country>
        <region>Bordeaux</region>
      </origin>
10     <price>22.80</price>
     <year>2002</year>
    </wine>
    <wine name="Riesling Hugel" appellation="Alsace">
15     <is-red>>false</is-red>
     <origin>
        <country>France</country>
        <region>Alsace and East</region>
      </origin>
     <price>17.95</price>
20     <year>2002</year>
    </wine>
  </wine-list>

```

- ❶ Root element of a small *data-oriented* XML document.
- ❷ First wine element, with two attributes.
- ❸ An attribute with a string value.
- ❹ An element with a boolean value.
- ❺ An element with a numerical value.
- ❻ Second wine element with two attributes.

Example 10.36. [wineList.json] A small wine list in JSON

The same content as in Example 10.35, transformed in JSON, the same callout numbers in both listings correspond to the same elements.

```

1 {
  "wine-list": {
    "wine": [{
      "name": "Domaine de l'Île Margaux",
5      "appellation": "Bordeaux supérieur",
      "is-red": true,
      "origin": {
        "country": "France",
        "region": "Bordeaux"
      },
10     },
     "price": 22.80,
     "year": 2002
    }
  ]
}

```

```

    }, {
15     "name": "Riesling Hugel",
        "appellation": "Alsace",
        "is-red": false,
        "origin": {
20         "country": "France",
            "region": "Alsace and East"
        },
        "price": 17.95,
        "year": 2002
    }
]
25 }}

```

- ❶ Root element translated as a single key-value pair.
- ❷ All wines will be stored under a single key corresponding to an array of values.
- ❸ An attribute is transformed into a key-value pair with a string value within quotes.
- ❹ A boolean value is written without quotes.
- ❺ A numerical value is written without quotes.
- ❻ Second wine of the array.

The JSON version given above was produced by Example 10.37, an XSLT stylesheet for translating an XML file in JSON format. We will describe later some restrictions that should be put on an XML file so that its translation to JSON is meaningful and produces legal Javascript.

Example 10.37. [xml2json.xsl] XSLT stylesheet to convert an XML file into JSON.

```

1 <!DOCTYPE stylesheet [
  <!ENTITY cr "<xsl:text>
  </xsl:text>">
  ]>
5 <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:gl="http://www.iro.umontreal.ca/lapalme"
  version="2.0">
  <xsl:strip-space elements="*" />
10 <xsl:output omit-xml-declaration="yes" />

  <xsl:template match="/">
    <xsl:text>{</xsl:text><xsl:apply-templates>
      <xsl:with-param name="indent" select="0" />
15 </xsl:apply-templates><xsl:text>}</xsl:text>
  </xsl:template>

  <!-- end current line, indent $str by $nb spaces -->
  <xsl:template name="doIndent">
20 <xsl:param name="nb" as="xsd:integer" />
    <xsl:param name="str" as="xsd:string" />
    &cr;<xsl:value-of
      select="concat(string-join(for $i in 1 to $nb return ' ', ''), $str)" />
  </xsl:template>

```

```

25 <!-- produce a JavaScript String:
    if the parameter is a number or a boolean, output the string;
    otherwise put the string between quotes escaping quotes within it-->
<xsl:function name="gl:jsString"> ④
30 <xsl:param name="s"/>
    <xsl:value-of select="
        if (string(number($s))!='NaN' or matches($s,'true|false','i')) then $s
        else concat('&quot;',replace($s,'&quot;','\\&quot;'),'&quot;')
    "/>
35 </xsl:function>

<xsl:template name="outArray"> ⑤
    <xsl:param name="indent" as="xsd:integer"/>
    <xsl:call-template name="doIndent">
40 <xsl:with-param name="nb" select="$indent+3" as="xsd:integer"/>
    <xsl:with-param name="str"
        select="concat(gl:jsString(string(current-grouping-key())),
            ':[')"/>
    </xsl:call-template>
45 <!-- output its contents "recursively" -->
    <xsl:apply-templates select="current-group()">
        <xsl:with-param name="indent" select="$indent+6" as="xsd:integer"/>
        <xsl:with-param name="outputName" select="false()"/>
    </xsl:apply-templates>
50 <xsl:call-template name="doIndent">
    <xsl:with-param name="nb" select="$indent+3" as="xsd:integer"/>
    <xsl:with-param name="str" select="']'" />
    </xsl:call-template>
</xsl:template>

55 <!-- an attribute is its name followed by its text value -->
<xsl:template match="@*"><xsl:param name="indent" as="xsd:integer"/> ⑥
    <xsl:call-template name="doIndent">
        <xsl:with-param name="nb" select="$indent" as="xsd:integer"/>
60 <xsl:with-param name="str"
            select="concat(gl:jsString(name()),': ',gl:jsString(.))"/>
    </xsl:call-template>
</xsl:template>

65 <!-- a text node is the value of its content --> ⑦
<xsl:template match="text()">
    <xsl:value-of select="gl:jsString(normalize-space())"/>
</xsl:template>

70 <!-- any child node --> ⑧
<xsl:template match="*">
    <xsl:param name="indent" as="xsd:integer"/>
    <xsl:param name="outputName" select="true()"/>
    <!-- output the name of the tag if so desired -->
75 <xsl:if test="$outputName"> ⑨
        <xsl:call-template name="doIndent">
            <xsl:with-param name="nb" select="$indent"/>
            <xsl:with-param name="str" select="gl:jsString(name())"/>

```

```

    </xsl:call-template>
80    <xsl:text>: </xsl:text>
</xsl:if>
<xsl:choose>
<!-- node with a single child text node which must be non-empty-->
<xsl:when test="count(text()[string-length(normalize-space(.))>0])=1">
85    <xsl:apply-templates> ❶
        <xsl:with-param name="indent" select="$indent+3"/>
    </xsl:apply-templates>
</xsl:when>
<!-- any type of nodes -->
90 <xsl:otherwise>
    <xsl:text>{</xsl:text>
    <!-- loops over all groups of identical tags --> ❷
    <xsl:for-each-group select="@*|*|text()"
        group-adjacent="local-name()"
95    <xsl:choose>
        <xsl:when test="count(current-group())=1">
            <!-- a single element group-->
            <xsl:apply-templates select=".">
                <xsl:with-param name="indent" select="$indent+3"/>
100        </xsl:apply-templates>
            </xsl:when>
            <xsl:otherwise>
                <!-- many element groups within braces-->
                <xsl:call-template name="outArray">
105                    <xsl:with-param name="indent" select="$indent"/>
                </xsl:call-template>
            </xsl:otherwise>
        </xsl:choose>
        <xsl:if test="position()=last()">
110            <xsl:text>,</xsl:text>
        </xsl:if>
    </xsl:for-each-group>
    <!-- end current group -->
    <xsl:call-template name="doIndent">
115        <xsl:with-param name="nb" select="$indent"/>
        <xsl:with-param name="str" select="'}'"/>
    </xsl:call-template>
</xsl:otherwise>
</xsl:choose>
120 <xsl:if test="position()=last()">
    <xsl:text>,</xsl:text>
</xsl:if>
</xsl:template>

125 </xsl:stylesheet>

```

- ❶ Defines an entity for ending a line with a carriage return.
- ❷ Starts the transformation of all node with an indentation of 0.

- ③ Named template for ending the current line and outputting a string indented with a given number of spaces.
- ④ Outputs a number, a boolean or a string in the correct Javascript format. It ensures that quotes are escaped within a string.
- ⑤ Outputs an array of similarly named elements. The name comes from the current grouping key found on line 92-⑩. An opening square bracket is output. The internal elements are converted recursively but by indicating that their name is not to be output. The closing square bracket is finally output.
- ⑥ An attribute is converted like a simple element: its name, a colon and the string value.
- ⑦ A text node is output but it is first normalized.
- ⑧ An element is output as a key-value pair.
- ⑨ If the name is to be output (it is the case unless we are dealing with an element within an array), the name is output properly indented, followed by a colon.
- ⑩ If it is a non-empty text node, output it properly indented.
- ⑪ When it is a complex node, group adjacent similarly named element within an array. When a group has only one element, then output it recursively properly indented. If there are many elements in the group, then use `outArray` line 37-⑥ for its elements. Separate each group with a comma and end with a closing brace.

Example 10.37 shows that it is relatively straightforward to produce JSON output from an XML file. The resulting Javascript string is trivially transformed into a Javascript data structure using the `eval` function. The browser then does not need a full blown XML parser to access the information from the XML structure.

But one should be aware that there are intrinsic limitations to the JSON notation compared to XML. It is not possible to do validation on the JSON structure as it is possible on XML using a Schema. Some JSON proponents argue that JSON is *typed* because the notation differentiates between numbers, strings, arrays and sets of key-value pairs while XML is *untyped* because it is only string. This is the case only when this is considered from the point of view of Javascript without any notion of XML embedded processing. In our view, XML is typed because it can be validated with a Schema but JSON is not...

Transforming XML elements into sets of key-value pairs also loses some information that might be important:

- the original document order is lost except within arrays that arise from a series of adjacent similarly named elements. In Javascript, key-value pairs within an object are not ordered, they behave like a hash table.
- there is no distinction between an attribute of an element and a children element as both are translated similarly.
- there is no defined translation for XML mixed content, i.e. a text node interspersed with other elements. In particular, Example 10.37 will produce illegal Javascript when it encounters mixed content because strings will be output within a key-value pair.

It is also possible to write an XSL stylesheet to transform JSON text into XML. The companion Web site of this document, gives one example in the file

```
json2xml.xsl
```

The processing is similar to the one used in Example 9.2 but doing this with a stylesheet implies a recursive processing of the textual input.

Because of the limitations given above, an XML transformed by Example 10.37 and then back with `json2xml.xsl` will not necessarily be identical: elements might be reordered within a node and an attribute of an element will be transformed into a child node of the element.

10.7.2. YAML

Another data structure notation that has been proposed is YAML[90], which could have meant *Yet Another Markup Language*, but is instead the recursive acronym of *YAML Ain't a Markup Language*. As one of the primary intended use of YAML is for configuration files, it was designed to be easy to read and write by humans using standard text editors. This contrasts with XML for which brevity was *not* a design goal because it was intended to be produced and parsed by machines.

Similarly to JSON, YAML describes tree structured data. The label of the root of the tree is written as a string line followed by colon. If the content is a simple value (string, number or boolean) then it is written on the same line. If the content is a complex value (hash table or array), it is written on the lines that follow, more deeply indented than the start of label of the root.

Example 10.38 shows how the content of our small wine list (Example 10.35) could be written in YAML. The same restrictions (no distinctions between attributes and nodes, no order of nodes, except within arrays) that we have described for JSON also apply to YAML. Parsers for YAML have been written for many programming languages so it means that it is possible to use it for exchanging data between systems written in different languages.

Example 10.38. [wineList.yaml] YAML version of Example 10.35

In order to show the parallel between XML, JSON and YAML, we use the same callout numbers as for Example 10.36.

```
1 wine-list:                               ❶
  wine:                                     ❷
  -
    name: "Domaine de l'Île Margaux"       ❸
5    appellation: Bordeaux supérieur      ❹
    is-red: true
    origin:
      country: France
      region: Bordeaux
10   price: 22.80                           ❺
    year: 2002
  -                                         ❻
    name: Riesling Hugel
15   appellation: Alsace
    is-red: false
    origin:
      region: Alsace and East
      country: France
20   price: 17.95
    year: 2002
```

- ❶ Root element on a single line with no indentation.
- ❷ Dependent element with an indentation that controls the next lines. The - on the following line indicates the start of an array element. The content of each array element is indented more deeply than the dash. A single value content is put on the same line.

- ③ A key with a string value. A string is not put within quotes unless it includes some special characters like quotes or newlines.
- ④ A key with a boolean value.
- ⑤ A key with a numerical value.
- ⑥ Start of the second wine of the array.

This section has shown *simpler* alternatives for markup languages for describing tree structured information. In special cases (configuration files, simple information exchanges), they provide enough structuring to be useful but in the more general set-up in which document ordering is important, they do not allow the full expression power of XML.

10.8. Additional information on alternative approaches

The Ruby Web site [81] is the primary source for information about Ruby and its implementation on most platforms. But the authoritative book is the *Pick Ax* book [44] and the Ruby-doc Web site [82]. The REXML Web site [80] describes XML processing in Ruby.

The Python programming book [32] is an excellent source of information for learning the language. This book hints at some XML processing in the *Advanced Internet Topics* chapter. As Python now integrates XML processing within its software package, the most up to date information about the use of Python for XML is the section of the documentation on structured markup processing [40].

PHP is described extensively on the web [76] and the XML modules are described in a section of the online manual [77].

JavaScript [68] was originally designed to add animation to web pages, but it has since become the *lingua franca* for all web applications in browsers, but also on the server side.

Swift [85] was introduced in 2015 originally for programming applications for the Apple machines, but it was then released as an open source project and has been ported to other platforms.

Chapter 18 of *Essential ActionScript 3.0* [39] presents a well organized introduction to both XML programming and how to use E4X in ActionScript 3.0.

Chapter 11. Conclusion

This report has presented some XML techniques using a single example application in order to give a pedagogical overview of the different approaches to processing XML files. The fact that we could apply different processing models on the same example illustrates the richness of the XML world which is part of almost every facet of computer systems. Working on this example was our own way of learning XML, so the techniques we used in this document should not be viewed as optimal or definitive. We also tried to make some connections with other computer science techniques in order to profit from our previous knowledge.

Even though, XML has been jokingly defined as *Lisp with fat parentheses*, we think that we have shown that there is much more to it in the sense of type checking, programming and interoperability and ease of use by means of many public domain efficient tools. Given the breadth of applications and the multitude of competing proposals to extend or use XML, we have deliberately ignored many details in order to focus on the main ideas, which are at end quite simple.

XML is now a given fact of life in computer science often in hidden ways (e.g. with the AJAX (Asynchronous Javascript And XML) technology that is now part of every modern web browser which also include a stylesheet interpreter; XML files can now be sent directly without having to be translated into HTML before hand. XML processing is only starting and much more remains to be done, especially because it is one of the fundamental building block of the Semantic Web initiative [8]. XML is the encoding for upper level languages such as RDF for defining information about documents and for OWL to define ontologies.

Bibliography

Documents

- [1] Adobe Corporation, *Portable Document Format*, Technical Report, 2005, http://partners.adobe.com/public/developer/pdf/index_reference.html
- [2] Dean Allemang and Jim Hendler, *Semantic Web for the Working Ontologist (2nd ed)*, Morgan Kaufman, 2011.
- [3] Altova Corporation, *XMLSpy 2007*, 2007, <http://www.altova.com/>
- [4] Apache XML Project, *Xerces Java and C++ Parsers*, 2005, <http://xerces.apache.org/xerces2-j/>
- [5] E. Armstrong, *The J2EE 1.4 Tutorial for Sun Java System Application Server Platform Edition 8.1 2005Q2*, June 2005, <http://docs.oracle.com/javaee/1.4/tutorial/doc/>
- [6] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon, *XML Path Language (XPath) 2.0*, Technical Report, 2010, <http://www.w3.org/TR/xpath20/>
- [7] Anders Berglund, *Extensible Stylesheet Language (XSL) Version 1.1*, 2006, <http://www.w3.org/TR/xsl/>
- [8] Tim Berners-Lee, James Hendler, and Ora Lassila, *The Semantic Web*, Scientific American, May 2001, <http://www.scientificamerican.com/article.cfm?id=the-semantic-web>
- [9] Paul V. Biron and Ashok Malhotra, *XML Schema Part 2: Datatypes Second Edition*, Technical Report, 2004, <http://www.w3.org/TR/xmlschema-2/>
- [10] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon, *XQuery 1.0: An XML Query Language*, Technical Report, 2010, <http://www.w3.org/TR/xquery/>
- [11] Bert Bos, Tantek Çelik, Ian Hickson, and Hakon Wium Lie, *Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification*, 2007, <http://www.w3.org/TR/CSS21/>
- [12] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau, *Extensible Markup Language (XML) 1.0 (Fourth Edition)*, Technical Report, 2006, <http://www.w3.org/TR/REC-xml/>
- [13] Allen Brown, Matthew Fuchs, Jonathan Robie, and Philip Wadler, *XML Schema: Formal Description*, Technical Report, September 2001, <http://www.w3.org/TR/xmlschema-formal/>
- [14] Alison Cawsey, *Presenting tailored resource descriptions: Will XSLT do the job?*, May 2000, <http://www9.org/w9cdrom/119/119.html>
- [15] James Clark and Makoto Murata, *RELAX NG Specification*, Technical Report, 2001, <https://www.oasis-open.org/committees/relax-ng/spec-20011203.html>
- [16] James Clark, *Multi-Format Schema Converter Based on RELAX NG*, Technical Report, 2003, <http://www.thaiopensource.com/relaxng/trang.html>
- [17] James Clark and Steve DeRose, *XML Path Language (XPath)*, Technical Report, 1999, <http://www.w3.org/TR/xpath/>
- [18] James Clark, *Associating Style Sheets with XML documents*, Technical Report, 1999, <http://www.w3.org/TR/xml-stylesheet/>
- [19] James Clark, *New mode for XML*, 2004, <http://www.thaiopensource.com/nxml-mode/>

- [20] Roger Costello, *Tutorials on Schematron - Rule-Based XML Validation*, 2007, <http://www.xfront.com/schematron/>
- [21] ECM Systemintegration, *XML4cobol SE*, Technical Report, 2005, <http://xml4cobol.com/>
- [22] David C. Fallside and Priscilla Walmsey, *XML Schema Part 0: Primer Second Edition*, Technical Report, 2004, <http://www.w3.org/TR/xmlschema-0/>
- [23] Benoît Habert, *Objectif : CLOS*, Masson, Paris, 1996.
- [24] Pascal Hitzler, Markus Krötzsch, and Sebastian Rudolph, *Foundations of Semantic Web Technologies*, Chapman & Hall / CRC, 2009, http://www.semantic-web-book.org/page/Foundations_of_Semantic_Web_Technologies
- [25] Cay Horstmann, *Big Java 3rd Edition*, 2008, <http://www.horstmann.com/bigjava.html>
- [26] ISO (International Organization for Standardization). ISO 8879:1986(E), *Information processing, Text and Office Systems Standard Generalized Markup Language (SGML). First edition*, Technical Report, 1986.
- [27] Michael Kay, *XSLT 2.0 and XPath 2.0, 4e ed.*, Wiley, 2008.
- [28] Michael Kay, *XSL Transformations (XSLT) Version 2.0*, 2007, <http://www.w3.org/TR/xslt>
- [29] Dongwon Lee and Wesley W. Chu, *Comparative Analysis of Six XML Schema Languages*, 2000, <http://www.cobase.cs.ucla.edu/tech-docs/dongwon/sigmod-record-00.pdf>
- [30] Linked Data, *Linked Data - Connect Distributed Data across the Web*, <http://linkeddata.org/>
- [31] Doug Lovell, *XSL Formatting Objects Developer's Handbook*, Sams, 2003.
- [32] Mark Lutz, *Programming Python 4th edition*, O'Reilly, 2010, <http://shop.oreilly.com/product/mobile/9780596158118.do?green=18055703844&cmp=af-mybuy-9780596158118.IP>
- [33] Ashok Malhotra, Jim Melton, and Norman Walsh, *XQuery 1.0 and XPath 2.0 Functions and Operators*, 2007, <http://www.w3.org/TR/xpath-functions/>
- [34] Murali Mani and Dongwon Lee, *XML to Relational Conversion using Theory of Regular Tree Grammars*, August 2002, <http://www.cobase.cs.ucla.edu/tech-docs/dongwon/eextt02.pdf>
- [35] Jonathan Marsh and David Orchard, *XML Inclusions (XInclude) Version 1.0*, Technical Report, 2004, <http://www.w3.org/TR/xinclude/>
- [36] Brett McLaughlin and Justin Edelson, *Java & XML, Third Edition*, 2006, <http://shop.oreilly.com/product/mobile/9780596101497.do>
- [37] W. Scott Means and Elliotte Rusty Harold, *XML in a Nutshell*, 2002, <http://shop.oreilly.com/product/mobile/9780596002923.do>
- [38] David Megginson, *Official website for SAX*, Technical Report, 2005, <http://www.saxproject.org/>
- [39] Colin Moock, *Essential ActionScript 3.0*, O'Reilly, 2007.
- [40] Python Software Foundation, *Structured Markup Processing Tools*, 2010, <http://docs.python.org/library/markup.html>
- [41] Leo Sauermann and Richard Cyganiak, *Cool URIs for the Semantic Web*, 2008, <http://www.w3.org/TR/cooluris/>
- [42] Matt Sergeant, *XML-Parser-2.34*, Technical Report, 2003, <http://search.cpan.org/~msergeant/XML-Parser-2.34/Parser.pm>

-
- [43] Bob Stayton, *DocBook XSL, The Complete Guide*, Sagehill Enterprises, 2005.
- [44] Dave Thomas, *Programming Ruby*, The Pragmatic Programmers, 2006.
- [45] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn, *XML Schema Part 1: Structures Second Edition*, Technical Report, 2004, <http://www.w3.org/TR/xmlschema-1/>
- [46] Henry S. Thompson and Richard Tobin, *XML Schema Validator*, 2002, <http://www.ltg.ed.ac.uk/~ht/xsv-status.html>
- [47] W3C, *Document Object Model (DOM) Technical Reports*, Technical Report, 2003, <http://www.w3.org/DOM/DOMTR>
- [48] W3C, *Universal Resource Identifiers*, Technical Report, 2003, http://www.w3.org/Addressing/URL/URI_Overview.html
- [49] W3C, *Uniform Resource Locators*, Technical Report, 2003, <http://www.w3.org/Addressing/URL/Overview.html>
- [50] Malcom Wallace and Colin Runciman, *HaXml*, 2006, <http://www.cs.york.ac.uk/fp/HaXml/>
- [51] Norman Walsh and Leonard Mueller, *DocBook 5.0: The Definitive Guide*, 2007, <http://www.docbook.org/tdg5/en/html/docbook.html>
- [52] Sean Wheller, *<oXygen/> XML Editor User Guide*, 2005, <http://www.oxygenxml.com/>
- [53] Jan Wielemaker, *SWI-Prolog SGML/XML parser*, Technical Report, 2005, <http://www.swi-prolog.org/pldoc/package/sgml.html>
- [54] Wikibook, *XQuery Examples Collection Wikibook!*, <http://en.wikibooks.org/wiki/XQuery>
- [55] Wikibook, *XQuery/SPARQL Tutorial*, http://en.wikibooks.org/wiki/XQuery/SPARQL_Tutorial
- [56] Graham Wilcock, *Pipelines, Templates and Transformations: XML and Natural Language Generation*, 2001, <http://www.ling.helsinki.fi/~gwilcock/Pubs/2001/NLPXML-01.pdf>
- [57] Richard York, *Beginning CSS - Cascading Style Sheets for Web Design*, Wiley, 2005.
- [58] Liyang Yu, *A Developer's Guide to the Semantic Web*, Springer, 2011.
- [59] Eric van der Vlist, *RELAX NG*, O'Reilly, 2004, <http://books.xmlschemata.org/relaxng/>
- [60] Eric van der Vlist, *XML Schema*, O'Reilly, 2002.
- [61] Priscilla Walmsley, *XQuery*, O'Reilly, 2002.

Web Sites

- [62] *The XML C parser and toolkit of Gnome*, <http://xmlsoft.org/>
- [63] *System.Xml*, <http://msdn.microsoft.com/en-us/library/system.xml.aspx>
- [64] *libxml++*, <http://libxmlplusplus.sourceforge.net/>
- [65] *eXist-db Open Source Native XML Database*, <http://exist-db.org/>
- [66] *Standard ECMA-357 - ECMAScript for XML (E4X) Specification*, ISO/IEC 22537, <http://www.ecma-international.org/publications/standards/Ecma-357.htm>

- [67] *Information technology - Document Schema Definition Languages (DSDL) - Part 3: Rule-based validation - Schematron*, [http://standards.iso.org/ittf/PubliclyAvailableStandards/c040833_ISO_IEC_19757-3_2006\(E\).zip](http://standards.iso.org/ittf/PubliclyAvailableStandards/c040833_ISO_IEC_19757-3_2006(E).zip)
- [68] *The Modern JavaScript Tutorial*, <https://javascript.info>
- [69] *A Semantic Web Framework for Java*, <http://jena.apache.org/>
- [70] *Pure-JavaScript implementation of the DOM and HTML Standards for use with Node.js*, <https://www.npmjs.com/package/jsdom>
- [71] *JavaScript Object Notation*, <http://www.json.org/>
- [72] *The Java Web Services Tutorial*, http://docs.oracle.com/cd/E17802_01/webservices/webservices/docs/1.6/tutorial/doc/
- [73] *The XML C parser and toolkit of Gnome*, <http://xmlsoft.org/>
- [74] *LIBX**, <http://www.explain.com.au/libx/>
- [75] *OWL 2 Web Ontology Language*, <http://www.w3.org/TR/owl-overview/>
- [76] *PHP Manual*, <http://php.net/>
- [77] *PHP Manual - XML Manipulation*, <http://php.net/manual/en/refs.xml.php>
- [78] *Resource Description Framework (RDF)*, <http://www.w3.org/RDF/>
- [79] *RDF Validation service*, <http://www.w3.org/RDF/Validator/>
- [80] *Ruby Electric XML*, <http://www.germane-software.com/software/XML/rexml/>
- [81] *Ruby A Programmer's Best Friend*, <http://www.ruby-lang.org/>
- [82] *RUBY-DOC.ORG - Help and documentation for the Ruby programming language.*, <http://www.ruby-doc.org/>
- [83] *XSLT 2.0 and XQuery processors*, <http://www.saxonica.com/documentation/about/intro.xml>
- [84] *SPARQL Query Language for RDF*, <http://www.w3.org/TR/rdf-sparql-query/>
- [85] *Swift programming language*, <https://swift.org>
- [86] *Schematron: a language for making assertions about patterns found in XML documents*, <http://www.schematron.com/>
- [87] *The Schematron Assertion Language 1.6*, <http://xml.ascc.net/resource/schematron/Schematron2000.html>
- [88] *Turtle - Terse RDF Triple Language*, <http://www.w3.org/TR/turtle/>
- [89] *Twinkle: A SPARQL Query Tool*, <http://www.ldodds.com/projects/twinkle/> Local improvement (description in French)
- [90] *YAML Ain't a Markup Language*, <http://www.yaml.org/>

Appendix A. Some XML Related Technologies and Systems

Abbreviation	Full name	Sections	<i>Specs</i>
DOM	Document Object Model	Section 8.1	[47]
DTD	Document Type Definition	Section 3.1	[12]
E4X	EcmaScript for XML	Section 10.6	[66]
JSON	Javascript Object Notation	Section 10.7.1	[71]
PDF	Portable Document Format	Section 5.4	[1]
RELAX NG	REGular LAnguage for XML, New Generation	Section 3.3	[15]
SAX	Simple Application programming interface for Xml	Section 8.2	[38]
SGML	Standard Generalized Markup Language	Chapter 1	[26]
URI	Uniform Resource Identifier	Section 2.1	[48]
URL	Uniform Resource Locator	Section 2.1	[49]
XML	eXtended Markup Language	*	[12]
XML Schema	XML Schema	Section 3.2	[45],[9]
XPath	XML PATH language	Chapter 4	[17]
XSL	eXtensible Stylesheet Language	Section 5.1	[7]
XSLT	XSL Transformations	Section 5.1	[28]
YAML	YAML Ai'nt a Markup Language	Section 10.7.2	[90]

Appendix B. Quick Reference Tables

Quick reference tables taken from previous chapters. Names in italics refer to other elements. The syntax of regular expressions used to describe the allowed forms are given in Table B.1. When there is an ambiguity between the symbols used for the regular expression syntax and their use as terminals of the grammar, the terminals are enclosed in chevrons (« »).

B.1. Regular expression

Table B.1. Regular expressions

,	sequence
	choice
()	grouping of expressions
?	optional previous expression
*	repetition, possibly none, of the previous expression
+	repetition at least once of the previous expression

B.2. DTD

Table B.2. Table 3.1, Section 3.1

<code><!DOCTYPE rootElement SYSTEM «"»file name«"» «>» {[!ENTITY *]}? «>»</code>
<code><!ELEMENT NCName «(» {#PCDATA « » }? regexPOf element name «)» «>»</code>
<code><!ELEMENT NCName (#PCDATA) «>»</code>
<code><!ELEMENT NCName EMPTY «>»</code>
<code><!ATTLIST elementNCName attributeNCName declValue default «>» declValue = CDATA ID IDREF «(» CNAME+ «)» default = {#REQUIRED #IMPLIED}</code>
<code><![CDATA[...]]«>»</code>
<code><!ENTITY name «"» ... «"» «>»</code>
<code><!ENTITY % name «"» ... «"» «>»</code>
<code><!ENTITY name SYSTEM «"»file name«"» «>»</code>

B.3. XML Schema

Table B.3. Table 3.2, Section 3.2

<pre><xs:schema targetNamespace="URI"> xs:import* {xs:simpleType xs:complexType xs:element xs:group}* </xs:schema></pre>
<pre><xs:import namespace="URI" schemaLocation="URI"/></pre>
<pre><xs:simpleType name="NCName"> xs:restriction </xs:simpleType></pre>
<pre><xs:complexType name="NCName" mixed="true"> {xs:choice xs:sequence xs:group}? xs:attribute* </xs:complexType></pre>
<pre><xs:element name="QName" type="TName"/></pre>
<pre><xs:element name="QName" ref="ENAME"/></pre>
<pre><xs:element name="QName"> {xs:simpleType xs:complexType}? {xs:unique xs:key xs:keyref}* </xs:element></pre>
<pre><xs:sequence {min max}occurs="nonNegativeInteger unbounded"> {xs:element xs:choice xs:sequence xs:group}* </xs:sequence></pre>
<pre><xs:choice {min max}occurs="nonNegativeInteger unbounded"> {xs:element xs:choice xs:sequence xs:group}* </xs:choice></pre>
<pre><xs:group name="NCName"> {xs:choice xs:sequence}* </xs:group></pre>
<pre><xs:attribute name="NCName" type="TName" use="required"/></pre>
<pre><xs:restriction base="TName"> <xs:{max min}{in/ex}clusive value="anySimpleType"/> <xs:{max min}length value="nonNegativeInteger" <pattern value="regExp" <enumeration value="anyValue" </xs:restriction></pre>
<pre><xs:{unique key} name="NCName"> xs:selector xs:field+ </xs:{unique key}></pre>
<pre><xs:keyref name="NCName" refer="NCName"> xs:selector xs:field+ </xs:keyref></pre>
<pre><xs:{selector field} xpath="XPathExpr"/></pre>

B.4. RELAX NG

Table B.4. Table 3.3, Section 3.3

Compact Syntax (RNC)	XML Syntax (RNG)
<pre>{default? namespace id=URI datatypes id=URI}* { start=pattern id=pattern }*</pre>	<pre><grammar> {<start> pattern </start> <define name="NCName">pattern+</define>}* </grammar></pre>
Patterns	
element QName «{» pattern «}»	<element name="QName">pattern+ </element>
attribute QName «{» pattern «}»	<attribute name="QName">pattern+ </attribute>
pattern{«,» pattern}+	<group name="QName">pattern+ </group>
pattern{«&» pattern}+	<interleave name="QName">pattern+ </interleave>
pattern{« » pattern}+	<choice name="QName">pattern+ </choice>
pattern«?»	<optional name="QName">pattern+ </optional>
pattern«*»	<zeroOrMore name="QName">pattern+ </zeroOrMore>
pattern«+»	<oneOrMore name="QName">pattern+ </oneOrMore>
mixed «{» pattern «}»	<mixed name="QName">pattern+ </mixed>
id	<ref name="NCName" />
empty	<empty/>
text	<text/>
data TypeValue	<value {name="NCName"}?>string+ </value>
data TypeValue «{»{id=value}* «}»	<data {type="NCName"}?> {<param name="NCName">string</param>}* </data>

B.5. Schematron

Table B.5. Table 3.4, Section 3.4

<pre><schema targetNamespace="URI"> title? ns? pattern+ </schema></pre>
<pre><ns prefix="QName" uri="URI"/></pre>
<pre><title> PCDATA </title></pre>
<pre><pattern abstract="yes no" id="ID" is-a="IDREF"> param* rule+ </pattern></pre>
<pre><param name="QName" value="XPathExpr"/></pre>
<pre><rule context="XPathExpr"> {let report assert}* </rule></pre>
<pre><let name="QName" value="XPathExpr"/></pre>
<pre><report test="XPathExpr"> {PCDATA value-of name}* </report></pre>
<pre><assert test="XPathExpr"> {PCDATA value-of name}* </assert></pre>
<pre><value-of select="XPathExpr"/></pre>
<pre><name/></pre>

B.6. XSLT

Table B.6. Table 5.1, Section 5.1

<pre><xsl:stylesheet> xsl:import*, (declaration/xsl:variable/xsl:param)* </xsl:stylesheet></pre>
<pre><xsl:template match="pattern" name="QName"> xsl:param*, sequence-constructor* </xsl:template></pre>
<pre><xsl:param name="QName" select="expression"> sequence-constructor </xsl:param></pre>
<pre><xsl:apply-templates select="expression"> (xsl:sort*/xsl:with-param)* </xsl:apply-templates></pre>
<pre><xsl:call-template name="Qname"/></pre>
<pre><xsl:with-param name="QName" select="expression"> sequence-constructor </xsl:with-param></pre>
<pre><xsl:function name="QName"> xsl:param*, sequence-constructor* </xsl:function></pre>
<pre><xsl:value-of select="expression"> sequence-constructor </xsl:value-of></pre>
<pre><xsl:variable name="QName" select="expression"> sequence-constructor </xsl:variable></pre>
<pre><xsl:copy> sequence-constructor </xsl:copy></pre>
<pre><xsl:copy-of select="expression"/></pre>
<pre><xsl:if test="expression"> sequence-constructor </xsl:if></pre>
<pre><choose> xsl:when*, xsl:otherwise? </choose></pre>
<pre><xsl:when test="expression"> sequence-constructor </xsl:when></pre>
<pre><xsl:otherwise> sequence-constructor </xsl:otherwise></pre>
<pre><xsl:for-each select="expression"> xsl:sort*, sequence-constructor </xsl:for-each></pre>
<pre><xsl:for-each-group select="expression" group-by="expression"> xsl:sort*, sequence-constructor </xsl:for-each-group></pre>

<pre><xsl:sort select="expression" data-type="{string}"> sequence-constructor </xsl:sort></pre>
<pre><xsl:key name="qname" match="pattern" use="expression"> sequence-constructor </xsl:key></pre>
<pre><xsl:element name="{string}"> sequence-constructor </xsl:element></pre>
<pre><xsl:text> character data </xsl:text></pre>
<pre><xsl:attribute name="{string}" select="expression"> sequence-constructor </xsl:attribute></pre>
<pre><xsl:attribute-set name="QName" use-attribute-sets="Qnames"> xsl:attribute* </xsl:attribute-set></pre>
<pre><xsl:message> sequence-constructor </xsl:message></pre>

Appendix C. XML Production of this Document

In this document, we have shown how XML related technologies are used in an applicative context: extraction, transformation and presentation of information. But one important application is the use of XML for storing and publishing texts; in fact, it was this type of application that motivated SGML, the ancestor of XML and that influenced many of the features we described in this document. This is why, we decided to also use XML technologies for organizing this document. In fact, the first version was written using LaTeX but it was later retrofitted into XML when we realized that we might as well practice what we preach given the fact that there are many excellent XML technologies for technical documentation publication.

This appendix briefly describes the organization of this document as a set of about 50 valid XML files (taking the example files into account) valid according to the DocBook 5.0 schema [51], written in RELAX NG Compact. A document of this size is inconvenient to manipulate as a whole, so we make an extensive use of `Xinclude` to combine chapters and sections written in separate files. The DocBook 5.0 RELAX NG Compact being the result of many years of experience, it is designed to be modular so it is possible to get the validation of each section and chapter separately. With modern XML editors, such as `oXygen/`, we can profit from real-time valid completions and validation which are very useful when writing a document. Using *validation scenarios*, it is even possible to validate references between different files within the global final document. The examples are `Xincluded` directly from the source XML used for testing. With the proper organization, that means that the examples are always up to date. Of course, making sure that the surrounding text still describes what is going on is still a hand made error prone process... We have also developed some special templates that include only subparts of examples in order to shorten the document while staying in parallel with the source programs.

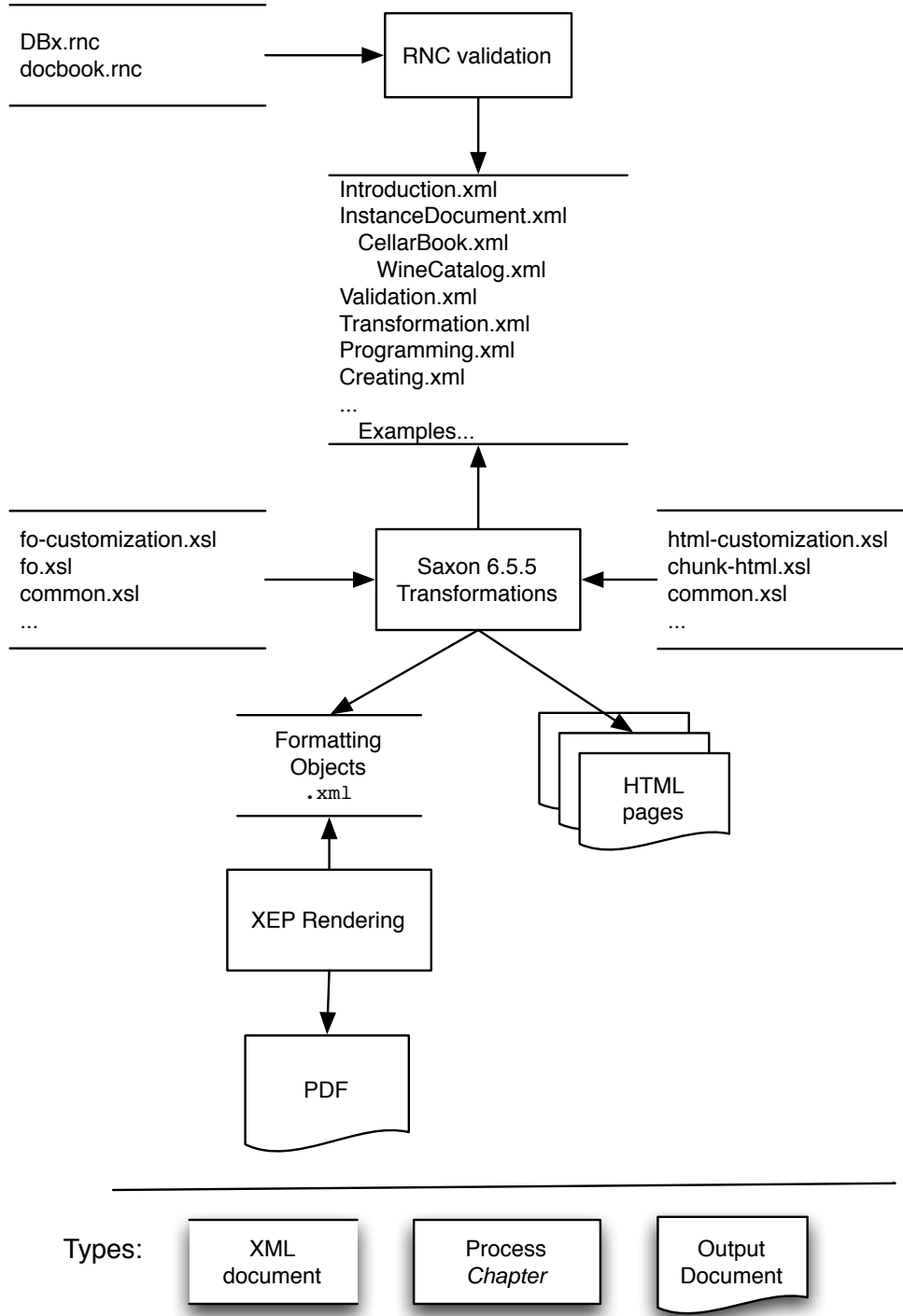
The publication process, shown in Figure C.1 is also done using some XML technologies presented in this document: from a single set of DocBook files, there are already stylesheets to transform them into either a single HTML file or, as we did, a set of related HTML files (a process called chunking in the DocBook jargon). But more, there also exist stylesheets to transform XML files into an XSL-FO file which is then rendered as a pdf file.

The DocBook 5.0 schema and stylesheets are already well developed but more interesting is the fact that they are designed to be extensible [43]. So it was possible for us to add a new XML tag (we called it `refline`) designed to indicate the line number within a listing. So in order that the XML file still be valid, we could extend the original schema incrementally by adding this new tag. We then also added a new template to implement the intended semantics to the original DocBook 5.0 stylesheets which are also designed to be customized either by means of parameters, definition of new templates or redefinition of existing templates.

Example C.1 shows the structure of most of the Docbook elements that we used when writing this document. It can serve as a reminder when writing a technical document to be processed by a Docbook stylesheet.

Figure C.1. Overview of organization of the DocBook XML source files of this document

We do not show here all the XML files involved but only the main organization. Inclusion of files is shown by indentation: an included file is more indented than its including file. This figure can be compared with Figure 1.3 (page 4).



Example C.1. [DBTemplates.xml] Examples of the most common uses of DocBook 5 elements in this document.

```
1 <?xml version="1.0" encoding="UTF-8"?>
  <!-- example uses of the most common docbook templates -->
  <?oxygen RNGSchema="DBx.rnc" type="compact"?>
  <!DOCTYPE book [
5 <!ENTITY % raliEnt SYSTEM "rali.ent">
      %raliEnt;
  ]>
  <chapter
    xmlns="http://docbook.org/ns/docbook"
10    xmlns:xi="http://www.w3.org/2001/XInclude"
    xmlns:xl="http://www.w3.org/1999/xlink"
    xml:id="chap_XXX">

    <title>Some DocBook 5 templates</title>
15
    <info>
      <keywordset><keyword>keyword for this document</keyword></keywordset>
    </info>

20    <indexterm class="startofrange" xml:id="section_start">
      <primary>instance document</primary>
    </indexterm>
    <!-- range of the section -->
    <indexterm class="endofrange" startref="section_start"/>
25
    <section xml:id="sec_title">
      <title>Title</title>
      <indexterm><primary>index term</primary></indexterm>

30    <!-- list : bulleted -->
    <itemizedlist spacing="compact">
      <listitem>
        <para>first bullet...</para>
      </listitem>
35      <listitem>
        <para>second bullet...</para>
      </listitem>
    </itemizedlist>

40    <!-- list : definition -->
    <variablelist>
      <?dbfo list-presentation="blocks"?> <!-- if terms are too long -->
      <varlistentry>
        <term>TERM</term>
45      <listitem><simpara>DEF</simpara></listitem>
      </varlistentry>
    </variablelist>
```

```
<!-- figure -->
50 <figure xml:id="fig_NAME" floatstyle="before">
    <title>TITLE</title>
    <simpara>CAPTION at the top</simpara>
    <mediaobject>
        <imageobject role="html"><!-- role are needed only for a pdf that do not appe
55         <imagedata fileref="images/WineTree.png" contentwidth="8in" width="8in"/>
        </imageobject>
        <imageobject role="fo">
            <imagedata fileref="images/WineTree.pdf" scale="75"/>
        </imageobject>
60    </mediaobject>
    <caption><simpara/></caption><!-- if a line is wanted at the bottom -->
</figure>
<!-- table -->

65 <table xml:id="tab_NAME" shortentry="1">
    <title>TITLE</title>
    <titleabbrev>TITLE small</titleabbrev>
    <?db-fo keep-together="always"?> <!-- to force a table to be on the same page -->
    <tgroup cols="2" align="left">
70        <tbody>
            <row>
                <entry>ROW1_COL1</entry>
                <entry>ROW1_COL2</entry>
            </row>
75            <row>
                <entry>ROW2_COL2</entry>
                <entry>ROW2_COL2</entry>
            </row>
        </tbody>
80    </tgroup>
    <caption><simpara>CAPTION</simpara></caption>
</table>

85 <!-- example listing -->
<example xml:id="fileName.ext">
    <title>[<link xl:href="&r-root;fileName.ext"><filename>fileName.ext</filename></l
    <indexterm>
        <primary>examples</primary>
90        <secondary>fileName.ext</secondary>
    </indexterm>
    <simpara>caption to appear at the example.</simpara>
    <!-- listing with callouts -->
    <programlistingco>
95        <areaspec units="linecolumn">
            <area xml:id="co_id" coords="L 70"/>
        </areaspec>
        <programlisting linenumbering="numbered">
            <!--<xi:include href="file..." parse="text"/>--></programlisting>
100        <calloutlist>
            <callout arearefs="co_id">
                <para>XXX</para>
            </callout>
        </calloutlist>
    </programlistingco>
</example>
```

```

        </callout>
        </calloutlist>
105    </programlistingco>
</example>

<para> <!-- things that can appear in a paragraph... -->
    <!-- references to line numbers in a listing (personal extension to docbook
110    <refline lineid="co_id"/>
    <!--<refline lineid="co_id" source="yes"/>-->
    <refline lineid="co_id" linkend="fileName.ext"/>

    <!-- italics and bold face -->
115    <emphasis>italics</emphasis>
    <emphasis role="strong">bold</emphasis>

    <!-- link to an external url -->
    <link xl:href="http:..." />
120    <link xl:href="http:...">TEXT</link>

    <!-- references to an ID,
        can also be used for referring to a bibliography entries -->
    <xref linkend="ID"/>
125    <xref linkend="ID" xrefstyle="select: label page"/>

    <!-- index terms -->
    <indexterm class="startofrange" xml:id="PRIMARY_start">
        <primary>PRIMARY</primary>
130    </indexterm>
    <indexterm class="endofrange" startref="PRIMARY_start"/>

    <indexterm><primary>PRIMARY</primary></indexterm>
</para>
135 </section>

<!-- forcing a page break or clearing floats-->
<?hard-pagebreak?>
<?float-clear?>

140 <!-- bibliography entry -->
<bibliography>
    <biblioentry xml:id="ID">
        <authorgroup>
            <author>
145                <personname>
                    <firstname>FIRSTNAME</firstname>
                    <surname>NAME</surname>
                </personname>
            </author>
150        </authorgroup>
        <publishername>PUB</publishername>
        <pubdate>YYYY</pubdate>
        <title>TITRE</title>
155    </biblioentry>
</bibliography>

```

</chapter>

Appendix D. Instance documents used in the examples

Example D.1. XML content of the file [CellarBook.xml] describing the cellar book whose structure has been given in Example 2.2

```
1 <cellar-book noNamespaceSchemaLocation="CellarBook.xsd">
  <wine-catalog>
    <wine name="Domaine de l'Île Margaux"
      appellation="Bordeaux supérieur"
5      classification="a.c."
      code="C00043125"
      format="750ml">
      <properties>
        <color>red</color>
10      <alcoholic-strength>12.5</alcoholic-strength>
        <nature>still</nature>
      </properties>
      <origin>
        <country>France</country>
15      <region>Bordeaux</region>
        <producer> SCEA Domaine de L'Île Margaux (B.P. 5) </producer>
      </origin>
      <comment>Ready for drinking now</comment>
      <food-pairing> Accompanies <emph>Bordelaise ribsteak</emph>,
20      <bold>pork with prunes</bold> or magret de canard. </food-pairing>
      <price>22.80</price>
      <year>2002</year>
    </wine>
    <wine name="Riesling Hugel"
25      appellation="Alsace"
      classification="a.c."
      code="C00042101"
      format="750ml">
      <properties>
        <color>white</color>
30      <alcoholic-strength>12</alcoholic-strength>
        <nature>still</nature>
      </properties>
      <origin>
        <country>France</country>
35      <region>Alsace and East</region>
        <producer>Hugel & Fils</producer>
      </origin>
      <price>17.95</price>
40      <year>2002</year>
    </wine>
    <wine name="Château Montguéret"
      appellation="Anjou"
      classification="a.c."
```

```
45         code="C00871996"
           format="750ml">
<properties>
  <color>rosé</color>
  <alcoholic-strength>11</alcoholic-strength>
50     <nature>still</nature>
</properties>
<origin>
  <country>France</country>
  <region>Loire Valley</region>
55     <producer>SCEA Château de Montguéret</producer>
</origin>
<comment> Made with Grolleau (100%). Ready to drink now. Serve at 8°-10°C. </
<tasting-note> Tender pink in color, this wine shows
  <emph>light raspberry</emph> highlights. </tasting-note>
60 <price>14.65</price>
  <year>2003</year>
</wine>
<wine name="Mumm Cordon Rouge"
  appellation="Champagne"
65   classification="a.c."
  code="C00312363"
  format="375ml">
<properties>
  <color>white</color>
70   <alcoholic-strength>12</alcoholic-strength>
  <nature>Champagne</nature>
</properties>
<origin>
  <country>France</country>
75   <region>Champagne</region>
  <producer>G.H. Martel & Co</producer>
</origin>
<comment> Ready for drinking now. Serve it fresh but not too cold. </comment>
<tasting-note> This champagne has a light fruity aroma. It is
80   delicate and has exquisite bubbles. </tasting-note>
  <price>33.00</price>
  <year>2000</year>
</wine>
<wine name="Prado Rey Roble"
85   appellation="Ribera-del-duero"
  classification="d.o."
  code="C00929026"
  format="magnum">
<properties>
90   <color>red</color>
  <alcoholic-strength>12.5</alcoholic-strength>
  <nature>still</nature>
</properties>
<origin>
95   <country>Spain</country>
  <region>Old Castille</region>
  <producer>Real Sitio de Ventosilla SA</producer>
</origin>
```

```

100         <price>35.25</price>
           <year>2002</year>
        </wine>
</wine-catalog>
<owner>
  <name>
105     <first>Jude</first>
        <family>Raisin</family>
        </name>
        <street>1234 rue des Châteaux</street>
        <city>St-George</city>
110     <province>ON</province>
        <postal-code>M7W 7S0</postal-code>
  </owner>
  <location>
115     <street>4587 des Futailles</street>
        <city>Vallée des crus</city>
        <province>QC</province>
        <postal-code>H3C 4J8</postal-code>
  </location>
  <cellar>
120     <wine code="C00043125">
        <purchaseDate>2005-06-20</purchaseDate>
        <quantity>2</quantity>
        <comment><cat:bold>Guy Lapalme, Montréal</cat:bold>: should reorder soo
125     </wine>
        <wine code="C00312363">
        <purchaseDate>2004-11-19</purchaseDate>
        <quantity>5</quantity>
        <rating stars="3"/>
        <comment>Bottle too small...</comment>
130     </wine>
        <wine code="C00871996">
        <purchaseDate>2005-06-19</purchaseDate>
        <quantity>0</quantity>
        <comment>Really great</comment>
135     </wine>
        <wine code="C00929026">
        <purchaseDate>2003-10-15</purchaseDate>
        <quantity>1</quantity>
        <comment>for <cat:bold>big</cat:bold> parties</comment>
140     </wine>
  </cellar>
</cellar-book>

```

This listing shows what is *seen* by the XML processor, after the inclusion of the wine catalog (see Example D.2) and once the XML entities have been replaced in Example 2.2. The formatting and line numbers differ from the ones in the source file.

Example D.2. XML content of the file [WineCatalog.xml] describing the wine catalog whose structure has been given in Example 2.3

```
1 <wine-catalog schemaLocation="http://www.iro.umontreal.ca/lapalme/wine-catalog WineCatalog
  <wine name="Domaine de l'Île Margaux"
    appellation="Bordeaux supérieur"
    classification="a.c."
5     code="C00043125"
      format="750ml">
  <properties>
    <color>red</color>
    <alcoholic-strength>12.5</alcoholic-strength>
10    <nature>still</nature>
  </properties>
  <origin>
    <country>France</country>
    <region>Bordeaux</region>
15    <producer>
      SCEA Domaine de L'Île Margaux (B.P. 5)
    </producer>
  </origin>
  <comment>Ready for drinking now</comment>
20  <food-pairing>
    Accompanies <emph>Bordelaise ribsteak</emph>,
    <bold>pork with prunes</bold> or magret de canard.
  </food-pairing>
  <price>22.80</price>
25  <year>2002</year>
</wine>
  <wine name="Riesling Hugel"
    appellation="Alsace"
    classification="a.c."
30    code="C00042101"
      format="750ml">
  <properties>
    <color>white</color>
    <alcoholic-strength>12</alcoholic-strength>
35    <nature>still</nature>
  </properties>
  <origin>
    <country>France</country>
    <region>Alsace and East</region>
40    <producer>Hugel & Fils</producer>
  </origin>
  <price>17.95</price>
  <year>2002</year>
</wine>
45  <wine name="Château Montguéret"
    appellation="Anjou"
    classification="a.c."
    code="C00871996"
    format="750ml">
50  <properties>
```

```

        <color>rosé</color>
        <alcoholic-strength>11</alcoholic-strength>
        <nature>still</nature>
55    </properties>
        <origin>
            <country>France</country>
            <region>Loire Valley</region>
            <producer>SCEA Château de Montguéret</producer>
60    </origin>
        <comment>
            Made with Grolleau (100%). Ready to drink now.
            Serve at 8°-10°C.
        </comment>
        <tasting-note>
65    Tender pink in color, this wine shows
            <emph>light raspberry</emph> highlights.
        </tasting-note>
        <price>14.65</price>
        <year>2003</year>
70    </wine>
        <wine name="Mumm Cordon Rouge"
            appellation="Champagne"
            classification="a.c."
            code="C00312363"
75    format="375ml">
        <properties>
            <color>white</color>
            <alcoholic-strength>12</alcoholic-strength>
            <nature>Champagne</nature>
80    </properties>
        <origin>
            <country>France</country>
            <region>Champagne</region>
            <producer>G.H. Martel & Co</producer>
85    </origin>
        <comment>
            Ready for drinking now. Serve it fresh but not too cold.
        </comment>
        <tasting-note>
90    This champagne has a light fruity aroma. It is delicate
            and has exquisite bubbles.
        </tasting-note>
        <price>33.00</price>
        <year>2000</year>
95    </wine>
        <wine name="Prado Rey Roble"
            appellation="Ribera-del-duero"
            classification="d.o."
            code="C00929026"
100   format="magnum">
        <properties>
            <color>red</color>
            <alcoholic-strength>12.5</alcoholic-strength>
            <nature>still</nature>

```

```
105     </properties>
        <origin>
            <country>Spain</country>
            <region>Old Castille</region>
            <producer>Real Sitio de Ventosilla SA</producer>
110     </origin>
        <price>35.25</price>
        <year>2002</year>
    </wine>
</wine-catalog>
115
```

This listing shows what is *seen* by the XML processor. The formatting and line numbers differ from the ones in the source file.

Index

A

AJAX, 93, 214, 223
algorithmic checking, 43
alternative notations for XML, 214
Alternatives to XML, 157
atomic values, XPath, 51
ATTLIST, 18
attribute, 2
attribute node type, 52
Attribute Value Template, 71
attributeFormDefault, 35
AVT, 71
axis specifier, 52
axis steps, XPath, 53
axis, XPath, 51-52

B

bottom-up schema organization, 22

C

cardinality checking, 43
Cascading Style Sheet, 93
command line
 query, 112
comment node type, 52
comments, 2
compact notation, 4
complex type, 34
constants, XPath, 52
cooccurrence checking, 43
CSS, 93-94

D

DocBook, xvii, 237
DOCTYPE, 11
document creation, 145
 DOM, 145-149
 E4X, 210
 JavaScript, 196
 PHP, 184
 Python, 171
 Ruby, 161
 Swift, 203
document nodes, XPath, 51
Document Object Model, 127-128

document order, 2
document parsing
 E4X, 209
 JavaScript, 194
 PHP, 187
 Python, 173
document parsing, DOM
 PHP, 178
 Python, 165
 Ruby, 158
 Swift, 199
document parsing, SAX
 PHP, 180
 Python, 166
 Ruby, 159
 Swift, 200
document parsing, StAX
 PHP, 183
 Python, 168
DOM, 127-132
 document creation, 145-149
 Java API, 144
 tree view, 139-141
DTD, 17
 !ATTLIST, 18
 !ELEMENT, 18
 !ENTITY, 18
 association, 21
 parameter entity, 19
 syntax table, 18

E

E4X, 157
 document creation, 210-213
 document parsing, 209-210
 document processing, 206-213
 examples, 208
Ecmascript for XML, 157
element node type, 52
elementFormDefault, 35
elements, 2
entity, 11, 18
 definition, 48
 parameter, 19
 predefined, 18
 system, 19
ENTITY, 18
examples

CBWC-RDF-S.rdf, 118
CBWC-RDF-S.rq, 125
CBWC-RDF-S.ttl, 121
Cellarbook.dtd, 19
CellarBook.rnc, 39
CellarBook.sch, 45
Cellarbook.xml, 10, 243
 including WineCatalog.xml, 9
CellarBook.xq, 104
CellarBook.xsd, 25
CellarBook.xsl, 71
compact.css, 93
compact.xq, 111
compact.xsl, 82
CompactErrorHandler.java, 131
compactFO.xsl, 89
CompactHandler.java, 134
CompactHandler.php, 181
CompactHandler.py, 167
CompactHandler.rb, 159
compactHTML.php, 179
compactHTML.xq, 108
compactHTML.xsl, 78
CompactReader.java, 150
CompactTokenizer.as, 210
CompactTokenizer.java, 146
CompactTokenizer.js, 196
CompactTokenizer.php, 184
CompactTokenizer.py, 171
CompactTokenizer.rb, 161
DBTemplates.xml, 239
DOMCompact.as, 209
DOMCompact.java, 128
DOMCompact.js, 194
DOMCompact.php, 178
DOMCompact.py, 165
DOMCompact.rb, 158
DOMCompact.swift, 199
DOMExpand.as, 212
DOMExpand.java, 147
DOMExpand.js, 197
DOMExpand.php, 186
DOMExpand.py, 172
DOMExpand.swift, 203
ETCompact.py, 174
ETExpand.py, 175
expand.rb, 162
JTreeHandler.java, 142
NamespaceExample.xml, 13
SAXCompact.java, 133
SAXCompact.php, 181
SAXCompact.py, 166
SAXCompact.rb, 159
SAXCompact.swift, 201
SAXExpand.java, 149
SimpleXMLCompact.php, 189
SimpleXMLExpand.php, 191
SimpleXMLPath.php, 188
StAXCompact.java, 137
StAXCompact.php, 183
StAXCompact.py, 170
StAXExpand.java, 153
TreeViewer.java, 140
WineCatalog.dtd, 20
WineCatalog.rnc, 41
WineCatalog.xml, 12, 246
WineCatalog.xq, 101
WineCatalog.xsd, 29
WineCatalog.xsl, 66
WineList.json, 215
WineList.xml, 215
WineList.yaml, 220
xml2json.xsl, 216
XMLStreamReader.py, 168
XSLcompact.php, 187

F
filter, XPath, 53
for expression, XPath, 53
function calls, XPath, 53
functions, XPath, 55

G
generalized tree, 1

H
HTML chunking, 237
HTML transformation, 63
 XQuery, 101

I
ID, 17
IDREF, 17
IMPLIED, 18
include, 48
instance document, 3, 9-15
instance document structure, 9

introduction, 1-7

J

Java

- appendChild(), 145
- createElement(), 145
- createTextNode(), 145
- DocumentImpl(), 145
- setAttribute(), 145

JavaScript, 157

- document creation, 196-198
- document parsing, 194-196
- document processing, 193-198

Javascript Object Notation (JSON), 157

JAXP, 127

js

- HTML, 194

JSON, 214-219

- conversion from XML, 216
- typing, 219

JSON (Javascript Object Notation), 157

JTree, 138

L

labeled tree, 7

LaTeX, 237

Lisp, 1

list, 34

N

named template, 60

namespace, 13, 34

- default, 15
- prefix, 13

namespace node type, 52

node test, 52

node test, XPath, 51-52

node types, 52

- attribute, 52
- comment, 52
- element, 52
- namespace, 52
- processing instruction, 52
- root, 52
- text, 52

nXML, 50

O

operators, XPath, 53

OWL, 223

<oxygen/>, 49

P

parameter entity, 19

PHP

- document creation, 184-187

- document parsing, 187-192

- document processing, 178-192

- DOM document parsing, 178-180

- SAX document parsing, 180-183

- StAX document parsing, 183-184

predicate expression, XPath, 51

predicate, XPath, 52

processing instruction node type, 52

Programming, 127-144

pull-parser, 127

Python

- document creation, 171-173

- document parsing, 173-177

- document processing, 165-177

- DOM document parsing, 165-166

- SAX document parsing, 166-168

- StAX document parsing, 168-171

Q

quantified expression, XPath, 53

query

- command line, 112

- RDF, 124

- Turtle, 124

- XML, 99

R

range, XPath, 53

RDF, 223

- query, 124

RDF Schema, 123

RDF versus XML, 125

RDF(S), 123

Relax NG, 38

- attribute, 38

- compact notation, 38

- element, 38

- empty, 38

- enumeration, 39

- interleave, 38
- pattern, 38-39
- Syntax table, 38
- text, 38
- REQUIRED, 18
- root node type, 52
- Ruby, 157
 - document creation, 161-164
 - document processing, 157-164
 - DOM document parsing, 158-159
 - SAX document parsing, 159-161
- russian doll schema organization, 22

S

- SAX, 127, 132-136
 - event handling, 134
 - events creation, 149-153
 - tree view, 141-143
- SAXON, 96, 112
- schema, 22
 - complex type, 34
 - mixed, 34
 - namespace, 34
 - organization, 22
 - Relax NG, 38-42
 - simple type, 33
 - Syntax table, 23
 - Trang converter, 38
 - type hierarchy, 33
 - xs:all, 24
 - xs:attribute, 24
 - xs:choice, 24, 34
 - xs:complexType, 24
 - xs:element, 24
 - xs:group, 24
 - xs:import, 24, 35
 - xs:key, 24
 - xs:keyref, 24
 - xs:restriction, 24
 - xs:sequence, 24, 34
 - xs:simpleType, 24
 - xs:unique, 24
- schema association, 48
- schema organization, 22
- schematron, 43
 - Syntax table, 45
- Schematron
 - assertion, 43

- report, 43
- rule, 43
- Semantic constraints, 43
- Semantic Web, 113-125
- sequence, XPath, 51
- SGML, 17, 237
- Simple API for XML, 132
- simple type, 33
 - list, 34
 - union, 34
- StAX, 127, 136-138
 - document creation, 153-155
 - tree view, 143
- steps, XPath, 51
- Stream API for XML, 136
- stylesheet association, 96
- Swift, 157
 - document creation, 203-205
 - document processing, 199-203
 - DOM document parsing, 199-200
 - SAX document parsing, 200-203

T

- tag
 - empty, 2
 - end, 2
 - start, 2
- targetNameSpace, 35
- template, named, 60
- text node type, 52
- top-down schema organization, 22
- Trang schema converter, 38
- transformation, 59-97
 - stylesheet, 59
 - stylesheet association, 96
 - XPath, 51
 - XSLT, 59
- Tree view
 - Programming, 138-143
- tree view
 - DOM, 139-141
 - SAX, 141-143
 - StAX, 143
- triples, 116
- Turtle, 116
 - query, 124
- type
 - complex, 22

simple, 22
typing
 JSON, 219

U
union, 34
URI, 15

V
validation, 17-50
 DTD, 17-21
 schema, 22-37
 schematron, 43-48
variables, XPath, 52

W
well-formed, 17
well-formedness, 9

X
XALAN, 96
XHR (XML HTTP Request), 214
xi:include, 48
XML
 alternative notations, 214-221
 alternatives, 157
 conversion to JSON, 216
XML alternative
 JSON, 214
 YAML, 220
XML text editor, 17
XMLHttpRequest, 214
xmlns, 15
XMLSpy, 49
XMLStreamWriter
 writeAttribute(..), 153
 writeCharacters(..), 153
 writeEndElement(), 153
 writeStartDocument, 153
 writeStartElement(..), 153
XPath, 51-58
 atomic values, 51
 axis, 51-52
 axis steps, 53
 constants, 52
 document nodes, 51
 examples, 51, 57
 filter, 53
 for expression, 53
 function calls, 53
 key, 62
 node test, 51-52
 node types, 52
 operators, 53
 predicate, 52
 predicate expression, 51
 quantified expression, 53
 range, 53
 sequence, 51
 steps, 51
 system functions, 55
 variables, 52
XQuery, 99-112
 compact transformation, 110-112
 HTML transformation, 101-110
XSL, 59
XSL predicate, 52
XSL transformations, 60
XSL-FO, 59, 84
 fo:layout-master-set, 87
 fo:list-item-body, 88
 fo:list-item-label, 88
 fo:page-sequence, 87
 fo:page-sequence-master, 87
 fo:simple-page-master, 87
xsl:function, 60
XSLT, 59-63
 Attribute Value Template, 71
 AVT, 71
 built-in templates, 75
 compact transformation, 80-83
 dynamic element creation, 71
 formatting objects, 84-92
 HTML transformation, 63-79
 named template, 60
 processing model, 60
 syntax table, 62
 template, named, 60
 xsl:apply-templates, 61
 xsl:attribute, 62
 xsl:attribute-set, 62
 xsl:call-template, 60
 xsl:choose, 61
 xsl:copy-of, 61
 xsl:element, 62
 xsl:for-each, 61
 xsl:for-each-group, 61

- xsl:function, 60
- xsl:if, 61
- xsl:key, 62
- xsl:message, 62
- xsl:otherwise, 61
- xsl:sort, 61
- xsl:template, 60
- xsl:text, 62
- xsl:value-of, 61
- xsl:variable, 61
- xsl:when, 61
- xsltproc, 96

Y

- YAML, 220-221
- YAML Ain't a Markup Language (YAML), 220
- Yet Another Markup Language (YAML), 220