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 April 14, 2019
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 April 14, 2019

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 E4X (EcmaScript for XML) 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 Sérafet, 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
   6.1. 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
B.3. XML Schema ................................................................. 226
B.4. RELAX NG ................................................................. 227
B.5. Schematron ................................................................. 228
B.6. XSLT ................................................................. 229
C. XML Production of this Document ......................... 231
D. Instance documents ......................................................... 237
Index .................................................................... 243
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. E4X expression examples ...................................................................................................... 194
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 ...................................................... 225
List of Examples

2.1. Outline of CellarBook.xml including WineCatalog.xml in a different namespace .......... 9
2.2. Excerpt of CellarBook.xml, the XML instance document holding the content of the cellar .... 10
2.3. Excerpt of WineCatalog.xml ................................................................................................. 12
2.4. [NamespaceExample.xml] A simplistic example of declaration and use of namespaces .... 13
3.1. [CellarBook.dtd] DTD for the cellar book, validation of the instance file in Example 2.2 ........ 19
3.2. [WineCatalog.dtd] DTD to validate the wine catalog, included in Example 3.1 ................. 20
3.3. [CellarBook.xsd] XML Schema for the cellar book, validation of the instance file in Example 2.2 .... 21
3.4. [WineCatalog.xsd] XML Schema file to validate the instance document shown in Example 2.3. 25
3.5. Outline of CellarBook.xsd importing Example 3.4 in a different namespace. ...................... 29
3.6. [CellarBook.rnc] RELAX NG Compact schema for the cellar book to validate Example 2.2. 36
3.7. [WineCatalog.rnc] RELAX NG Compact schema for the wine catalog to validate Example 2.3 39
3.8. [CellarBook.sch] ISO-Schematron for the cellar book to validate Example 2.2. ................. 41
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. [WineCatalog.xsl] 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. [CellarBook.xsl] XSLT stylesheet to produce information about the cellar ...................... 71
5.5. [built-in-template-rules.xsl] 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. [compactHTML.xsl] 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. [compact.xsl]: Stylesheet used to transform Example 2.2 into Example 5.8 ..................... 82
5.10. [compactFO.xsl] Transformation of the cellar book into a colored nested blocks representa-
        tion ........................................................................................................................................ 89
5.11. [compact.css] for displaying the content of the cellar book ............................................. 93
6.1. [WineCatalog.xq] XQuery script to select the red wines in the catalog .............................. 101
6.2. [CellarBook.xq] XQuery script to produce information about the cellar ......................... 104
6.3. [compactHTML.xq] XQuery transformation to produce a bulleted list from the cellar book .... 108
6.4. [compact.xq]: Stylesheet used to transform Example 2.2 into Example 5.8 ..................... 111
7.1. [CBWC-RDF-S.rdf] Subject, Predicate and Object triples for the cellar book in RDF/XML. 118
7.2. [CBWC-RDF-S.ttl] The Turtle version of Example 7.1 ......................................................... 121
7.3. [CBWC-RDF-S.rq] SPARQL queries on Example 7.2 ......................................................... 125
8.1. [DOMCompact.java] Text compaction of the cellar book with Java using the DOM model .... 128
8.2. [CompactErrorHandler.java] Error handler of the DOM parsing of Example 8.1 .......... 131
8.3. [SAXCompact.java] Text compaction of the cellar book with Java using the SAX model .... 133
8.4. [CompactHandler.java] SAX handler for text compacting an XML file such as Example 2.2 ............................................................................................................................... 134
8.5. [StAXCompact.java] Text compaction of the cellar book with Java using the SAX model .... 137
8.6. [Treeviewer.java]: JTree building with DOM and StAX Processing of an XML file ......... 140
8.7. [JTreeHandler.java]: JTree building with SAX Processing of an XML file .................... 142
XML: Looking at the Forest Instead of the Trees

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 parsing 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
10.2. [SAXCompact.rb] Text compaction of the cellar book with Ruby using the SAX model .......... 159
10.3. [CompactHandler.rb] Ruby SAX handler for text compaction Example 2.2 ................................. 160
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 ........................................ 163
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 compaction Example 2.2 ............................. 167
10.10. [StAXCompact.py] Text compaction of the cellar book with Python using the StAX model .................. 168
10.11. [CompactTokenizer.py] Specialized string scanner that returns tokens of compact form .................. 170
10.12. [DOMExpand.py] Python compact form parsing to create an XML document ............................ 171
10.13. [ETCompact.py] Python compaction of an XML file using ElementTree nodes .................... 172
10.14. [ETExpand.py] Python compact form parsing to create an ElementTree document ............ 174
10.16. [compactHTML.php] HTML compaction of the cellar book with PHP using the DOM model ........... 179
10.18. [CompactHandler.php] PHP SAX handler for text compaction Example 2.2 ........... 181
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
10.23. [SimpleXMLPath.php] SimpleXML file loading followed by PHP SimpleXML expressions ........ 188
10.25. [SimpleXMLExpand.php] PHP compact form parsing to create a SimpleXML Element document .................................................. 191
10.27. [CompactTokenizer.as] E4X specialized string scanner that returns tokens of compact form .................. 196
10.28. [DOMExpand.as] E4X compact form parsing in order to create an XML document .................. 198
10.29. [DOMCompact.swift] Text compaction of the cellar book with Swift using the DOM model .......... 200
10.31. [DOMExpand.swift] Swift compact form parsing to create an XML document ..................... 204
10.32. [WineList.xml] A small wine list in XML ........................................................................ 208
10.33. [WineList.json] A small wine list in JSON .............................................................. 208
10.34. [xml2json.xsl] XSLT stylesheet to convert an XML file into JSON. ....................... 209
10.35. [WineList.yaml] YAML version of Example 10.32 ...................................................... 213
C.1. [DBTemplates.xml] Examples of the most common uses of DocBook 5 elements in this
document. ..................................................................................................................................... 233
D.1. XML content of the file describing the cellar book ....................................................... 237
D.2. XML content of the file describing the wine catalog ..................................................... 240
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
<?xml version="1.0" encoding="UTF-8"?>
<wine name="M" code="00518712" format="1l"> <!-- 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, <bold>Meatloaf,</bold> 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. "1l" or '1l'). 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 (soXygen/) 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="1L">
  <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>
```

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.
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
<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="1l"
    <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>
    </ul>
</li></ul>
</body>
</html>

- **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

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at [http://www.renderx.com/](http://www.renderx.com/)
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]]

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.
[The 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 ①) 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 9-①) described in an external file Wine-Catalog.xml
  - owner (line 11-①) name and address
  - location (line 21-①) address of the cellar (if different from that of the owner)
  - cellar (line 27-①) 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 WineCatalog.xsd"
   <wine name="Domaine de l'Île Margaux" ... >...</wine>
   <wine name="Riesling Hugel" ... >...</wine>
   <wine name="Château Montagu" ... >...</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>
```
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)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE cellar-book [<!ENTITY guy "Guy Lapalme" >
<!ENTITY eacute "&#xe9;" >
<!ENTITY mtl "Montréal" >
<!ENTITY GL "&guy;, &mtl;" ]>
<cellar-book xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:cat="http://www.iro.umontreal.ca/lapalme/wine-catalog"
xsi:noNamespaceSchemaLocation="CellarBook.xsd">
  <xi:include href="WineCatalog.xml" xmlns:xi="http://www.w3.org/2001/XInclude"/>
  <owner>
    <name>
      <first>Jude</first>
      <family>Raisin</family>
    </name>
    <street>1234 rue des Châteaux</street>
    <city>St-George</city>
    <province>ON</province>
    <postal-code>M7W 7S0</postal-code>
  </owner>
  <location>
    <street>4587 des Futailles</street>
    <city>Vallée des crus</city>
    <province>QC</province>
  </location>
  <cellar>
    <wine code="C00043125">
      <purchaseDate>2005-06-20</purchaseDate>
      <quantity>2</quantity>
      <comment>
        <cat:bold>&GL;</cat:bold>: should reorder soon
      </comment>
    </wine>
    ...<wine code="C00871996">
      <purchaseDate>2011-02-20</purchaseDate>
      <quantity>1</quantity>
      <comment>for <cat:bold>big</cat:bold> parties</comment>
    </wine>
    <wine code="C00929026">
      <purchaseDate>2003-10-15</purchaseDate>
      <quantity>1</quantity>
      <comment>
        <cat:bold>Montréal & Spain</cat:bold>: should reorder soon
      </comment>
    </wine>
  </cellar>
</cellar-book>
```

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.
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.

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.

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.

Starting tag indicating that this is an instance file to be validated with a given schema.

Name of the file containing the schema for this instance file.

Inclusion of another XML file describing the wine catalog.

Description of the owner of the cellar book.

Name of the owner of the cellar book.

Location of the owner of the cellar book.

Description of the cellar.

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.

As the bold element is defined in the schema associated with the catalog, it must be given the namespace prefix of the catalog.

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 website 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 &apos; and &quot; (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)

   WineCatalog.xsd"
xmlns="http://www.iro.umontreal.ca/lapalme/wine-catalog">
   <wine name="Domaine de l'Île Margaux" ... >
      <properties>
         <color>red</color>
         <alcoholic-strength>12.5</alcoholic-strength>
         <nature>still</nature>
      </properties>
      <origin>
         <country>France</country>
         <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>, <bold>pork with prunes</bold> or magret de canard.
      </food-pairing>
      <price>22.80</price>
      <year>2002</year>
   </wine>
   ...
   <wine name="Prado Rey Roble" ...
      <properties>
         <color>red</color>
         <alcoholic-strength>12.5</alcoholic-strength>
         <nature>still</nature>
      </properties>
      <origin>
         <country>Spain</country>
         <region>Old Castille</region>
         <producer>Real Sitio de Ventosilla SA</producer>
      </origin>
      <price>35.25</price>
      <year>2002</year>
   </wine>
</wine-catalog>

1 Start of the wine catalog description.
2 Namespace and file name of the Schema for this instance file.
3 Namespace for the wine catalog, useful to differentiate the wine elements that appear both in the catalog and the cellar book.
4 Description of a wine.
5 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 7-9). 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 18-9) 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 30-9), 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.

```xml
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">ghi</b>
       </a>
     </tree>
 10   <c xmlns="mary">ghi</c>
   </trees-with-namespaces>
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/
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 6-1 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 20-3) 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[ ... ]]> 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

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;!DOCTYPE rootElement SYSTEM &quot;file name&quot;</td>
<td>DOCTYPE declaration</td>
</tr>
<tr>
<td>&lt;!ELEMENT NCName &quot;</td>
<td>ELEMENT declaration</td>
</tr>
<tr>
<td>&lt;!ELEMENT NCName (#PCDATA)</td>
<td>ELEMENT declaration with PCDATA</td>
</tr>
<tr>
<td>&lt;!ELEMENT NCName EMPTY</td>
<td>ELEMENT declaration with EMPTY</td>
</tr>
<tr>
<td>&lt;!ATTLIST elementNCName attributeNCName declValue default</td>
<td>ATTLIST declaration</td>
</tr>
<tr>
<td>declValue = CDATA</td>
<td>Declaration values</td>
</tr>
<tr>
<td>default = #REQUIRED</td>
<td>Declaration default values</td>
</tr>
<tr>
<td>&lt;![CDATA[ ... ]]</td>
<td>CDATA declaration</td>
</tr>
<tr>
<td>&lt;!ENTITY name &quot;&quot;</td>
<td>ENTITY declaration</td>
</tr>
<tr>
<td>&lt;!ENTITY % name &quot;&quot;</td>
<td>Predefined entity declaration</td>
</tr>
<tr>
<td>&lt;!ENTITY name SYSTEM &quot;file name&quot;</td>
<td>SYSTEM entity declaration</td>
</tr>
</tbody>
</table>

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. 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 5-9 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 6-9) can contain character data and no other elements. A cellar (line 3-9) is a list (possibly empty) of wine elements. A name (line 12-9) 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 10-9).

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:

&lt; a less-than sign (<) in an XML file because < is reserved to indicate the start of a tag;

&amp; As we have introduced the ampersand to indicate the start of an entity, we need a way to insert an ampersand (&);

&quot; for inserting a ";

&amp;apos; for inserting a ' ;
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 26-6). Its call is preceded by a percent sign instead of an ampersand see owner (line 32-4) and location (line 33-6). Another special kind of entity, introduced by SYSTEM, refers to a file such as wine-catalog (line 36-6). 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\"?>
2  <!ELEMENT cellar (wine)* >
3  <!ELEMENT wine (purchaseDate,quantity,rating?,comment?) >
4         <!ELEMENT purchaseDate (#PCDATA) >
5         <!ELEMENT quantity (#PCDATA) >
6         <!ELEMENT rating EMPTY >
7         <!ATTLIST rating stars CDATA #IMPLIED>
8         <!ATTLIST wine code IDREF #REQUIRED >
9  <!ELEMENT name (first | family | initial)+ >
10       <!ELEMENT first (#PCDATA) >
11       <!ELEMENT family (#PCDATA) >
12       <!ELEMENT initial (#PCDATA) >
13  <!ELEMENT cellar-book (wine-catalog, owner, location, cellar) >
14  <!-- [general] entities for use in instance document -->
15       <!ENTITY guy "Guy Lapalme" >
16       <!ENTITY eacute "\&xe9;" >
17       <!ENTITY mtl "Montr\&eacute;al" >
18       <!ENTITY GL "&guy;, &mtl;" >
19  <!-- parameter entities for use within a DTD -->
20       <!ENTITY % address "(street, city, province, postal-code)" >
21         <!ELEMENT street (#PCDATA) >
22         <!ELEMENT city (#PCDATA) >
23         <!ELEMENT province (#PCDATA) >
24       <!ELEMENT postal-code (#PCDATA) >
25       <!ELEMENT owner (name,%address;) >
```
A cellar is a possibly empty list of wines.
A wine is defined by a purchase date, a quantity, an optional rating and an optional comment.
The purchase date is a string.
A quantity is also a string.
A code must refer to an id defined in the wine catalog.
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.
The cellar book is composed of the wine catalog, the owner, the location, and the cellar itself.
Some definitions of entities and how they can be composed.
Parameter entities can be used in the definition of other entities.
Use of a parameterized entity defined on line 26-
Another use of the same parameterized entity.
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-
The attribute format (line 10- 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- of type ID and its reference in line 10- 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,
   (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 >
<!ATTLIST cat-wine format (375ml | 750ml | l | magnum
   | jeroboam | rehoboam | mathusaleh | salmanazar
   | balthazar | nabuchodonosor) #REQUIRED >
<!ATTLIST cat-wine code ID #REQUIRED>
<!ELEMENT properties (color,alcoholic-strength,nature?) >
<!ELEMENT color (#PCDATA) >
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.

The format is given as a list of literal values from which the actual value of the attribute must be chosen.

The unique code for the wine that must be matched by a reference at line 10-1 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:

```xml
<!DOCTYPE cellar-book SYSTEM "CellarBook.dtd">
<cellar-book>
  <wine-catalog>
    ...
  </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

<table>
<thead>
<tr>
<th>&lt;xs:schema targetNamespace=&quot;URI&quot;&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>xs:import* {xs:simpleType</td>
</tr>
<tr>
<td>&lt;/xs:schema&gt;</td>
</tr>
<tr>
<td>&lt;xs:import nameSpace=&quot;URI&quot;</td>
</tr>
<tr>
<td>schemaLocation=&quot;URI&quot;/&gt;</td>
</tr>
<tr>
<td>&lt;/xs:schema&gt;</td>
</tr>
<tr>
<td>&lt;xs:simpleType name=&quot;NCName&quot;</td>
</tr>
<tr>
<td>xs:restriction</td>
</tr>
<tr>
<td>&lt;/xs:simpleType&gt;</td>
</tr>
<tr>
<td>&lt;xs:complexType name=&quot;NCName&quot;</td>
</tr>
<tr>
<td>mixed=&quot;true&quot;&gt;</td>
</tr>
<tr>
<td>{xs:choice</td>
</tr>
<tr>
<td>&lt;/xs:complexType&gt;</td>
</tr>
<tr>
<td>&lt;xs:element name=&quot;QName&quot;</td>
</tr>
<tr>
<td>type=&quot;TName&quot;/&gt;</td>
</tr>
<tr>
<td>&lt;/xs:element&gt;</td>
</tr>
<tr>
<td>&lt;xs:element name=&quot;QName&quot;</td>
</tr>
<tr>
<td>ref=&quot;EName&quot;/&gt;</td>
</tr>
<tr>
<td>&lt;/xs:element&gt;</td>
</tr>
<tr>
<td>&lt;xs:element name=&quot;QName&quot;&gt;</td>
</tr>
<tr>
<td>{xs:simpleType</td>
</tr>
<tr>
<td>{xs:unique</td>
</tr>
<tr>
<td>&lt;/xs:element&gt;</td>
</tr>
<tr>
<td>&lt;xs:sequence {min</td>
</tr>
<tr>
<td>{xs:element</td>
</tr>
<tr>
<td>&lt;/xs:sequence&gt;</td>
</tr>
<tr>
<td>&lt;xs:choice {min</td>
</tr>
<tr>
<td>{xs:element</td>
</tr>
<tr>
<td>&lt;/xs:choice&gt;</td>
</tr>
<tr>
<td>&lt;xs:group name=&quot;NCName&quot;&gt;</td>
</tr>
<tr>
<td>{xs:choice</td>
</tr>
<tr>
<td>&lt;/xs:group&gt;</td>
</tr>
<tr>
<td>&lt;xs:attribute name=&quot;NCName&quot;</td>
</tr>
<tr>
<td>type=&quot;TName&quot;</td>
</tr>
<tr>
<td>use=&quot;required&quot;/&gt;</td>
</tr>
<tr>
<td>&lt;/xs:attribute&gt;</td>
</tr>
<tr>
<td>&lt;xs:restriction base=&quot;TName&quot;&gt;</td>
</tr>
<tr>
<td>&lt;xs:{max</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>&lt;/xs:restriction&gt;</td>
</tr>
<tr>
<td>&lt;xs:{unique</td>
</tr>
<tr>
<td>xs:selector xs:field+</td>
</tr>
<tr>
<td>&lt;/xs:{unique</td>
</tr>
<tr>
<td>&lt;xs:keyref name=&quot;NCName&quot;</td>
</tr>
<tr>
<td>refer=&quot;NCName&quot;&gt;</td>
</tr>
<tr>
<td>xs:selector xs:field+</td>
</tr>
<tr>
<td>&lt;/xs:keyref&gt;</td>
</tr>
<tr>
<td>&lt;xs:{selector</td>
</tr>
</tbody>
</table>

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

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

    <xs:complexType name="Wine">
        <xs:sequence>
            <xs:element name="purchaseDate" type="xs:date"/>
            <xs:element name="quantity" type="xs:nonNegativeInteger"/>
            <xs:element name="rating" minOccurs="0">
                <xs:complexType>
                    <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>

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

    <xs:element name="cellar-book">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="cat:wine-catalog"/>
                <xs:element name="owner" type="Owner"/>
                <xs:element name="location" minOccurs="0">
                    <xs:complexType>
                        <xs:group ref="address"/>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
Gets the types defined in the schema for the wine catalog, which uses a different namespace, the one identified by the cat prefix.

Element cellar is a possibly empty list of wine elements of type Wine defined at line line 16-①. Definition of element wine of type Wine.

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.

The purchase date is of the predefined XML type date.
The rating is an empty element with an attribute `<stars>` indicating the number of stars, which must be a number greater than 0.

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.

The element `<name>` is a list of elements comprising the first name, the family name and initial in any order.

Each component of a name is a string.

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 of Example 3.1.

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 of Example 3.

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 and in the `<Owner>` type (line 84 of Example 3).

A postal code is a string matching a regular expression defined in the `<pattern>` element in the simple type defined at line 91 of Example 3.

The province is indicated by a two-letter code chosen among a predefined list.

An owner is a `<name>` element followed by the content of the `<address>` group, defined at line 57 of Example 3.

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.

```xml
1 <?xml version="1.0" encoding="UTF-8"?>
    <xs:schema 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">  

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

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

    </xs:complexType>
```

29
<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>

<xs:complexType name="Wine">
    <xs:sequence>
        <xs:element name="properties" type="cat:Properties"/>
        <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" 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 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"/>
    <xs:attribute name="code" type="xs:ID"/>
    <xs:attribute name="format" type="cat:Format"/>
</xs:complexType>

<xs:complexType name="Properties">
    <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>
</xs:complexType>

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

<xs:simpleType name="Format">
    <xs:restriction base="xs:string">
        <xs:enumeration value="375ml"/>
        <xs:enumeration value="750ml"/>
        <xs:enumeration value="1l"/>
        <xs:enumeration value="magnum">
            <xs:annotation>
                <xs:documentation>1.5 litres</xs:documentation>
            </xs:annotation>
        </xs:enumeration>
    </xs:restriction>
</xs:simpleType>
<xs:annotation>
  <xs:documentation>3 litres</xs:documentation>
</xs:annotation>
</xs:enumeration>

<xs:enumeration value="rehoboam">
  <xs:annotation>
    <xs:documentation>4.5 litres</xs:documentation>
  </xs:annotation>
</xs:enumeration>

<xs:enumeration value="mathusalem">
  <xs:annotation>
    <xs:documentation>6 litres</xs:documentation>
  </xs:annotation>
</xs:enumeration>

<xs:enumeration value="salmanazar">
  <xs:annotation>
    <xs:documentation>9 litres</xs:documentation>
  </xs:annotation>
</xs:enumeration>

<xs:enumeration value="balthazar">
  <xs:annotation>
    <xs:documentation>12 litres</xs:documentation>
  </xs:annotation>
</xs:enumeration>

<xs:enumeration value="nabuchodonosor">
  <xs:annotation>
    <xs:documentation>15 litres</xs:documentation>
  </xs:annotation>
</xs:enumeration>

</xs:restriction>
</xs:simpleType>

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

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

<xs:simpleType name="Percentage">
  <xs:restriction base="xs:decimal">
  </xs:restriction>
</xs:simpleType>
Start of the schema for the wine catalog giving \textit{xs} prefix for the elements used in the definition of other elements.

If a schema has a target namespace such as in line 6-\textcircled{4}, \textit{qualified} means that all defined elements in this schema will be created in the target namespace.

\textit{unqualified} means that all attributes in this schema will belong to the same namespace as their carrying element.

Defines the namespace that will be used for the elements defined by this schema.

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 \textit{xml:base} attribute is added by the import process. This implies that this attribute must also be allowed in the schema.

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.

A wine in the catalog is assigned a unique code of type \textit{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.

Properties defines the color of the wine, its alcoholic content as a percentage defined by a type on line 120-\textcircled{4}. Finally, an optional nature can be given as a string.

\textit{Origin} is defined by the country, the region and the producer.

\textit{Format} is a string value taken within the values appearing in an \textit{xs:enumeration} element within the \textit{xs:restriction} element. The content of the \textit{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.

The \textit{mixed} attribute having a \textit{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 \textit{emph} and \textit{bold} elements.

\textit{color} can be only be one of three most often encountered possibilities. Wine connoisseurs might be \textit{shocked} that wines cannot take other colors such as \textit{yellow}.

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 22-1) which must not only be an integer but a positive one, first (line 34-1) 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 72-1) or to match a regular expression (PostalCodeCA on line 91-1).

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 8-1), or by indicating the type with the type attribute such as in wine (line 11-1) or purchaseDate (line 18-1).

A complex type is defined either by a sequence of elements contained in an xs:sequence element (see cellar on line 8-1) or by a choice between many elements contained in an xs:choice element such as within name (line 31-1). 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 27-1).

xs:choice and xs:sequence can be nested. For example, name (line 31-1) 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 84-1). 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 103-1) 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-). The same namespace is also assigned to the prefix cat.elementFormDefault is set to qualified (line 3- 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- 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-) and the wine-catalogelement (line 43-).

These examples show another interesting use of namespaces: to make sure that relative references are kept, an xml:base attribute (line 19-) is added to the root element (line 14-) 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- of Example 3.5).
Example 3.5. Outline of CellarBook.xsd importing Example 3.4 (shown in italics) in a different namespace.

```xml
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'
   xmlns:cat='http://www.iro.umontreal.ca/lapalme/wine-catalog'
   elementFormDefault="qualified"
   attributeFormDefault="unqualified"
   targetNamespace='http://www.iro.umontreal.ca/lapalme/wine-catalog'>
              schemaLocation=http://www.w3.org/2001/03/xml.xsd'/>
  <xs:element name='wine-catalog'>
    <xs:complexType>
      <xs:sequence minOccurs='0' maxOccurs='unbounded'>
        <xs:element name='wine' type='cat:Wine'/>
      </xs:sequence>
      <xs:attribute ref='xml:base'/>
    </xs:complexType>
  </xs:element>

  <xs:complexType name='Wine'>...
  </xs:complexType>

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

  <xs:complexType name='Wine'>...
  </xs:complexType>

  <xs:element name='cellar-book'>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref='cat:wine-catalog'/>
        <xs:element name='owner' type='Owner'/>
        <xs:element name='location' minOccurs='0'>...
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```
Overview of the XML Schemas

50     ...
     </xs:schema>

1. The target namespace of the imported schema is the one specified by the `cat` prefix.
2. Imports another schema whose content is shown here in italics.
3. The wine catalog is a list of wine descriptions.
4. Ensures that the `base` attribute added by the import process is properly defined.
5. Type for the wine description in the catalog.
6. Definition of the cellar in the top-level schema.
7. `wine` elements in the cellar refer to the `Wine` type definition of the cellar.
8. Type for the wine description in the cellar.
9. Reference to the imported type for the wine code.
10. Top-level element of the cellar.
11. Reference to the imported type for the wine-catalog (line 14-①).
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 8-1, line 18-1 and line 24-1). A definition can also be a comma-separated sequence of patterns (line 31-1 and line 37-1). 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

<table>
<thead>
<tr>
<th>Compact Syntax (RNC)</th>
<th>XML Syntax (RNG)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;grammar</code></td>
<td><code>&lt;grammar&gt;</code></td>
</tr>
<tr>
<td>`{default=namespace id=URI</td>
<td><code>&lt;start&gt;</code> pattern &lt;/start&gt;`</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><code>}</code>*</td>
<td><code>}</code>*</td>
</tr>
<tr>
<td><code>&lt;default? namespace id=URI</code></td>
<td><code>&lt;grammar&gt;</code></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;start=pattern</code></td>
<td><code>&lt;grammar&gt;</code></td>
</tr>
<tr>
<td><code>id=pattern</code></td>
<td><code>}</code>*</td>
</tr>
<tr>
<td><code>&lt;attribute</code></td>
<td><code>&lt;grammar&gt;</code></td>
</tr>
<tr>
<td><code>name=&quot;QName&quot;&gt;pattern&lt;/attribute&gt;</code></td>
<td><code>}</code>*</td>
</tr>
<tr>
<td><code>attribute</code></td>
<td><code>&lt;grammar&gt;</code></td>
</tr>
<tr>
<td><code>QName «{» pattern «}»</code></td>
<td><code>}</code>*</td>
</tr>
<tr>
<td><code>element</code></td>
<td><code>&lt;grammar&gt;</code></td>
</tr>
<tr>
<td><code>QName «{» pattern «}»</code></td>
<td><code>}</code>*</td>
</tr>
<tr>
<td><code>pattern{«,» pattern}+</code></td>
<td><code>&lt;grammar&gt;</code></td>
</tr>
<tr>
<td><code>attribute</code></td>
<td><code>&lt;grammar&gt;</code></td>
</tr>
<tr>
<td><code>QName «{» pattern «}»</code></td>
<td><code>}</code>*</td>
</tr>
<tr>
<td><code>pattern{«&amp;» pattern}+</code></td>
<td><code>&lt;grammar&gt;</code></td>
</tr>
<tr>
<td><code>pattern{«?»</code></td>
<td><code>&lt;grammar&gt;</code></td>
</tr>
<tr>
<td><code>pattern{«*»</code></td>
<td><code>&lt;grammar&gt;</code></td>
</tr>
<tr>
<td><code>pattern{«+»</code></td>
<td><code>&lt;grammar&gt;</code></td>
</tr>
<tr>
<td><code>mixed «{» pattern «}»</code></td>
<td><code>}</code>*</td>
</tr>
<tr>
<td><code>id «{» pattern «}»</code></td>
<td><code>&lt;grammar&gt;</code></td>
</tr>
<tr>
<td><code>empty</code></td>
<td><code>&lt;grammar&gt;</code></td>
</tr>
<tr>
<td><code>text</code></td>
<td><code>&lt;grammar&gt;</code></td>
</tr>
<tr>
<td><code>data TypeValue «{» id=value}+ «}»</code></td>
<td><code>&lt;grammar&gt;</code></td>
</tr>
<tr>
<td><code>ref</code></td>
<td><code>&lt;grammar&gt;</code></td>
</tr>
<tr>
<td><code>id</code></td>
<td><code>&lt;grammar&gt;</code></td>
</tr>
<tr>
<td><code>text</code></td>
<td><code>&lt;grammar&gt;</code></td>
</tr>
<tr>
<td><code>data TypeValue</code></td>
<td><code>&lt;grammar&gt;</code></td>
</tr>
<tr>
<td><code>value {name=&quot;NCName&quot;&gt;string+&lt;/value&gt;</code></td>
<td><code>}</code>*</td>
</tr>
<tr>
<td><code>data TypeValue</code></td>
<td><code>&lt;grammar&gt;</code></td>
</tr>
<tr>
<td><code>{&lt;param name=&quot;NCName&quot;&gt;string&lt;/param&gt;</code></td>
<td><code>}</code>*</td>
</tr>
</tbody>
</table>

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, data TypeName 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 4-9) 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 1-9). 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.
Defines the namespace prefix used in the catalog.
Defines the namespace prefix corresponding to the standard XML datatypes.
Includes the schema for the wine catalog.
Defines the start symbol for this file by redefining the start symbol imported from the wine catalog file.
The cellar element is a possibly empty list of wine elements.
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.
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.

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 8-10. Owner and Address are themselves patterns.

Pattern for the street, the city name, a Canadian province name (the list of available values that can be defined directly).

A pattern combining an element and a pattern.

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 4-10). 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.

```xml
1 default namespace = "http://www.iro.umontreal.ca/lapalme/wine-catalog"
datatype 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*
 }

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

```

41
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.

A pattern defining all the information related to a given wine.

A pattern describing some properties of a wine.

Informations about the origin of a wine.

List of allowable values for wine format.

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.

List of allowable values for the color of wines.

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], [84] 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 [85], 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

```
<schema targetNameSpace="URI">
  title? ns? pattern+
</schema>

<ns prefix="QName"
  uri="URI"/>
<title>
  PCDATA
</title>
<pattern abstract="yes | no"
  id="ID"
  is-a="IDREF">
  param* rule+
</pattern>

<param name="QName"
  value="XPathExpr"/>
<rule context="XPathExpr">
  {let | report | assert}*
</rule>

<let name="QName"
  value="XPathExpr"/>

<report test="XPathExpr">
  {PCDATA | value-of | name}*
</report>

<assert test="XPathExpr">
  {PCDATA | value-of | name}*
</assert>

<value-of select="XPathExpr"/>
<name/>
```

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 1 <?xml version="1.0" encoding="UTF-8"?>
  <schema xmlns="http://purl.oclc.org/dsdl/schematron"  
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"  
     queryBinding="xslt2">
  5  <title>Validation using Schematron rules</title>  
  9  <ns prefix="cat"
           url="http://www.iro.umontreal.ca/lapalme/wine-catalog"/>
     <xsl:key name="colors" match="/cellar-book/cat:wine-catalog/cat:wine"
                use="cat:properties/cat:color"/>
 10  <pattern>
      <rule context="wine">
        <report test="rating/@stars>1 and not(comment)">
          There should be a comment for a wine with more than one star.
```
20  <pattern context="cellar">
    <let name="nbBottles" value="sum(wine/quantity)"/>
    <report test="nbBottles &lt; 10">
        Only <value-of select="nbBottles"/> bottles left in the cellar.
    </report>
    <!-- nb of bottles of each color in the cellar -->
    <let name="winesFromCellar" value="/cellar-book/cellar/wine"/>
    <let name="nbReds" value="sum($winesFromCellar[@code=key('colors','red')]/quantity)"/>
    <let name="nbWhites" value="sum($winesFromCellar[@code=key('colors','white')]/quantity)"/>
    <let name="nbRosés" value="sum($winesFromCellar[@code=key('colors','rosé')]/quantity)"/>
    <let name="nbColors" value="nbReds+nbWhites+nbRosés"/>
    <!-- check for a well balanced cellar!!! -->
    <assert test="nbReds$nbColors div 3">
        Not enough reds (<value-of select="nbReds"/> over <value-of select="nbColors"/> left in the cellar.
    </assert>
    <assert test="nbWhites$nbColors div 4">
        Not enough whites (<value-of select="nbWhites"/> over <value-of select="nbColors"/> left in the cellar.
    </assert>
    <assert test="nbRosés$nbColors div 4">
        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">
        Inconsistent count of bottles: total is <value-of select="nbBottles"/> 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>
</pattern>
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- for indexing for indexing

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.

Gives an informative title to a set of validation rules.

Defines the namespace prefix to be used within XPath expressions that refer to elements of the wine catalog (see line 8-). For implementation reasons, it is not enough, or even necessary, to define the namespace with a xmlns attribute.

Declares that wine elements of the catalog are to be indexed by their color, used in line 26-. 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.

A pattern that defines a rule implementing a mix of cooccurrence and algorithmic constraint within a cellar element.

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.

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.

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.

Defines an abstract pattern for generating two other patterns (line 66-, line 69-). 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.

Instantiates the abstract pattern defined at line 57- 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- 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- 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-), 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-) 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):

```xml
<?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.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-type 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 237), 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 real 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 `<?` 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] \[ XPath :: = Expr \]
XPath

ExprSingle ::= ForExpr
       | QuantifiedExpr
       | IfExpr
       | OrExpr

ForExpr ::= SimpleForClause return ExprSingle

SimpleForClause ::= for $ VarName in ExprSingle ( , $ VarName in ExprSingle)*

QuantifiedExpr ::= (some | every) $ VarName in ExprSingle
                   ( , $ VarName in ExprSingle)* satisfies ExprSingle

IfExpr ::= if « (« Expr « ) » then ExprSingle else ExprSingle

OrExpr ::= AndExpr ( or AndExpr )*

AndExpr ::= ComparisonExpr ( and ComparisonExpr )*

ComparisonExpr ::= RangeExpr ( ( ValueComp | GeneralComp | NodeComp ) RangeExpr )?

RangeExpr ::= AdditiveExpr ( to AdditiveExpr )?

AdditiveExpr ::= MultiplicativeExpr ( « ( + | - ) MultiplicativeExpr » )

MultiplicativeExpr ::= UnionExpr ( ( « ( * | div | idiv | mod ) UnionExpr » )

UnionExpr ::= IntersectExceptExpr ( ( union | « | » ) IntersectExceptExpr )

IntersectExceptExpr ::= InstanceofExpr ( ( intersect | except ) InstanceofExpr )

InstanceofExpr ::= TreatExpr ( instance of SequenceType )?

TreatExpr ::= CastableExpr ( treat as SequenceType )?

CastableExpr ::= CastExpr ( castable as SingleType )?

CastExpr ::= UnaryExpr ( cast as SingleType )?

UnaryExpr ::= ( « | « » ) ValueExpr

ValueExpr ::= PathExpr

GeneralComp ::= « | != | < | <= | > | >=

ValueComp ::= eq | ne | lt | le | gt | ge

NodeComp ::= is | << | »>

PathExpr ::= (/ RelativePathExpr?)
       | ( // RelativePathExpr)
       | RelativePathExpr

RelativePathExpr ::= StepExpr (/ | //) StepExpr*

StepExpr ::= FilterExpr | AxisStep

AxisStep ::= ( ReverseStep | ForwardStep ) PredicateList

ForwardStep ::= ( ForwardAxis NodeTest ) | AbbrevForwardStep

ForwardAxis ::= ( child :: ) | ( descendant :: ) | ( attribute :: ) | ( self :: )
                | ( descendant-or-self :: ) | ( following-sibling :: )
                | ( following :: ) | ( namespace :: )

AbbrevForwardStep ::= @? NodeTest

ReverseStep ::= ( ReverseAxis NodeTest ) | AbbrevReverseStep

ReverseAxis ::= ( parent :: ) | ( ancestor :: ) | ( preceding-sibling :: )
               | ( preceding :: ) | ( ancestor-or-self :: )

AbbrevReverseStep ::= ..

NodeTest ::= KindTest | NameTest

NameTest ::= QName | Wildcard

Wildcard ::= « * » | ( NCName : « * » ) | ( « * » : NCName)

FilterExpr ::= PrimaryExpr PredicateList

PredicateList ::= Predicate*

Predicate ::= [ Expr ]

PrimaryExpr ::= Literal | VarRef | ParenthesizedExpr | ContextItemExpr | FunctionCall

Literal ::= NumericLiteral | StringLiteral
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

<table>
<thead>
<tr>
<th>Category</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Document information</strong></td>
<td>document-uri(node)</td>
<td>a string describing the URI of node</td>
</tr>
<tr>
<td><strong>Mathematics</strong></td>
<td>abs(numeric)</td>
<td>absolute value of numeric</td>
</tr>
<tr>
<td></td>
<td>ceiling(numeric)</td>
<td>smallest integer greater than numeric</td>
</tr>
<tr>
<td></td>
<td>floor(numeric)</td>
<td>biggest integer smaller than numeric</td>
</tr>
</tbody>
</table>
Strings
concat(arg₁, arg₂ ...)
string-join(strings, sep)
substring(string, start, length)
string-length(string)
normalize-space(string)
translate(string, mapStr, transStr)
lower-case upper-case (string)
contains(string1, string2)
starts-with(string1, string2)
ends-with(string1, string2)
substring-before(string1, string2)
substring-after(string1, string2)
matches(string, pattern)
replace(string, pattern, replacement)
tokenize(string, pattern)

Sequences
distinct-values(seq)
remove(seq, position)
reverse(seq)
subsequence(seq, start, length)
count(seq)
avg | max | min | sum(seq)

Context
position()
last()
current-date | current-time()
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 237)**

```
/cellar-book/owner  
/cellar-book/cellar/wine[quantity<2]  
/cellar-book/cellar/wine[1]  
//postal-code/..  
/cellar-book/owner/street  
//wine/@code  
//cat:wine/@codea  
//wine[last()]/@code  
//cellar-book/cellar/wine[1]/comment/cat:bold  
sum(/cellar-book/cellar/wine/quantity)  
for $w in //wine return  
  concat($w/quantity,':',//cat:wine/@code[.=.]/../@name)  
//cat:wine[cat:origin/cat:country='France' and cat:price<20]  
```

1. The **owner** element of the cellar. Result: node at line 103.
2. 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 `&lt;` (even within strings!). Result: nodes at lines 131 and 136.
3. The first wine of the cellar. Result: node at line 120.
4. 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.
5. The street of the cellar’s owner. Result: 1234 rue des Châteaux
6. The value of the **code** attribute for all wines in the cellar. Result: C00043125, C00312363, C10263859, C00929026.
7. 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.
8. The code of the last wine of the cellar. 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. Returns Guy Lapalme, Montréal (expanded from the entity &GL;)
10. 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.
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: 2:Domaine de l'île Margaux, 5:Mumm Cordon Rouge, 6:Château Montaguèret, 1:Prado Rey Roble.
12. 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:

```xml
<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 declare-
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 a **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 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 \texttt{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 \texttt{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 \texttt{xsl:element}, \texttt{xsl:attribute} and \texttt{xsl:text} elements. \texttt{xsl:attribute-set} is useful for grouping many attributes under a single name and to combine them.

\texttt{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 \textit{formatting objects} to produce more complex formatting.

\textbf{Table 5.1. XSLT syntax reminder}

\begin{verbatim}
<xsl:stylesheet>
  \xsl:import*, (declaration|\xsl:variable|\xsl:param)*
</xsl:stylesheet>

<xsl:template match="pattern" name="QName">
  \xsl:param*, sequence-constructor*
</xsl:template>

<xsl:param name="QName" select="expression">
  sequence-constructor
</xsl:param>

<xsl:apply-templates select="expression">
  (\xsl:sort*|\xsl:with-param)*
</xsl:apply-templates>

<xsl:call-template name="Qname"/>

<xsl:with-param name="QName" select="expression">
  sequence-constructor
</xsl:with-param>

<xsl:function name="QName">
  \xsl:param*, sequence-constructor*
</xsl:function>

<xsl:value-of select="expression">
  sequence-constructor
</xsl:value-of>

<xsl:variable name="QName" select="expression">
  sequence-constructor
</xsl:variable>

<xsl:copy>
  sequence-constructor
</xsl:copy>

<xsl:copy-of select="expression"/>
\end{verbatim}
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**

```
Wine Catalog (red only)

<table>
<thead>
<tr>
<th>Wine Name</th>
<th>Code</th>
<th>Color</th>
<th>Year</th>
<th>Price</th>
<th>ml</th>
<th>l</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domaine de l'Ile Margaux</td>
<td>C00043125</td>
<td>red</td>
<td>2002</td>
<td>$22.80</td>
<td>750</td>
<td>0.75</td>
</tr>
<tr>
<td>Prado Rey Roble</td>
<td>C00929026</td>
<td>red</td>
<td>2002</td>
<td>$35.25</td>
<td>1500</td>
<td>1.5</td>
</tr>
</tbody>
</table>
```

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

```xml
<?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/lopalme/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'Ile Margaux</td>
      <td>C00043125</td>
      <td>red</td>
      <td>2002</td>
      <td>$22.80</td>
      <td>750</td>
      <td>0.75</td>
    </tr>
    <tr>
      <td>Prado Rey Roble</td>
      <td>C00929026</td>
      <td>red</td>
      <td>2002</td>
      <td>$35.25</td>
      <td>1500</td>
      <td>1.5</td>
    </tr>
  </table>
</body>
</html>
```
The stylesheet in Example 5.2 first defines a template matching the root node (line 14-19). Because the wine catalog is defined in a specific namespace, its prefix must be declared (line 3-4) 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 23-28) outputs a global heading and then starts a table and defines its headers; the lines of the table will be filled by selecting (line 35-42) 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 6-9). 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 49-52) and a named template (line 52-55) 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

```xml
<?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"
    version="2.0">
    <xsl:param name="color" select="'red'"/>
    <!-- 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" encoding = "UTF-8"/>

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

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

<xsl:template match="cat:wine">
    <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="@year"/></td>
        <td align="right"><xsl:value-of select="format-number(cat:price,'$0.00')"/></td>
        <td align="right"><xsl:value-of select="cat:toML(@format)"/></td>
    </tr>
</xsl:template>
```
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.

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.

`xsl:output` declaration to tell the transformation engine to serialize with the appropriate headers to produce valid XHTML.

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).

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.

Uses a `for-each` in a sequence of strings to output the headers in the first row of the table.

Selects the wines with the chosen color. The result is a sequence `wine` nodes. The template on line 39- is applied to each of them.

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.
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**

<table>
<thead>
<tr>
<th>Personal address</th>
<th>Cellar address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234 rue des Châteaux St-George ON M7W 7S0</td>
<td>4587 des Futaillès Vallée des crus QC H3C 4J8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code</th>
<th>Name</th>
<th>Purchase Date</th>
<th>Rating</th>
<th>Nb bottles</th>
</tr>
</thead>
<tbody>
<tr>
<td>00043125</td>
<td><em>Domaine de l'Ile Margaux</em> <em>Domaine de l'Ile Margaux</em></td>
<td>2005-06-20</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>00312363</td>
<td><em>Mumm Cordon Rouge</em> <em>Mumm Cordon Rouge</em></td>
<td>2004-11-19</td>
<td>***</td>
<td>5</td>
</tr>
<tr>
<td>00871996</td>
<td><em>Château Montaguéret</em> <em>Château Montaguéret</em></td>
<td>2005-06-19</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>00929026</td>
<td><em>Prado Rey Roble</em> <em>Prado Rey Roble</em></td>
<td>2003-10-15</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**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
<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></tr>
      <tr>
        <td>...</td>
        <td>...</td>
      </tr>
    </table>
  </body>
</html>
```
Example 5.4 illustrates some new features. The root template (line 20-9) 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 12-9) 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 36-9 which creates a table with the address of the owner (line 43-9) and the cellar (line 47-9). It then calls the cellar template (line 54-9). The lines of the addresses are obtained by looping on all elements with a for-each and outputting a <br/> 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 43-9) with a position number greater than 1.

The cellar (line 58-9) 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 66-9) 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-ereço) to the total named template. To compute the total number of bottles (line 79-ereço), we can use the predefined sum function. Finally, if there are any comment in the wine elements of the cellar (line 84-ereço), 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-ereço) 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-ereço). 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-ereço) 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 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-ereço). 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-ereço). 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-ereço) 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-ereço) 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.
<!-- 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"
    encoding="UTF-8"/>

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

<!-- matches the root node -->
<xsl:template match="/"

<html>
<head>
<title>Cellar of <xsl:value-of select="cellar-book/owner/name"/>
</title>
</head>
<body>
<xsl:apply-templates/>
</body>
</html>
</xsl:template>

<!-- output of the content of the cellar: addresses of the owner and cellar followed by a table of wines -->
<xsl:template match="cellar-book"

<h1>Cellar of <xsl:value-of select="owner/name"/></h1>
<table border="1">
<tr>
<th>Personal address</th>
<th>Cellar address</th>
</tr>
<tr><td><xsl:for-each select="owner/*[position()>1]">
<xsl:apply-templates/><br/>
</xsl:for-each></td>
<td><xsl:for-each select="location/*">
<xsl:apply-templates/><br/>
</xsl:for-each></td>
</tr>
</table>
<p/>
<xsl:apply-templates select="cellar"/>
</xsl:template>

<!-- content of the cellar as a table -->
<xsl:template match="cellar"

<table border="1">
<tr><!-- head of the table -->
<xsl:for-each select="'Code','Name','Purchase Date','Rating','Nb bottles'">
</xsl:for-each>
</tr>
</table>
</xsl:template>
<table>
<thead>
<tr>
<th>#</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>65</td>
<td><code>&lt;th&gt;&lt;xsl:value-of select=&quot;.&quot;/&gt;&lt;/th&gt;</code></td>
</tr>
<tr>
<td>66</td>
<td><code>&lt;/xsl:for-each&gt;</code></td>
</tr>
<tr>
<td>67</td>
<td><code>&lt;/tr&gt;</code></td>
</tr>
<tr>
<td>68</td>
<td><code>&lt;xsl:apply-templates select=&quot;wine&quot;&gt;&lt;!-- each row --&gt;</code></td>
</tr>
<tr>
<td>69</td>
<td><code>&lt;xsl:sort select=&quot;@code&quot; order=&quot;ascending&quot;/&gt;</code></td>
</tr>
<tr>
<td>70</td>
<td><code>&lt;/xsl:apply-templates&gt;</code></td>
</tr>
<tr>
<td>71</td>
<td><code>&lt;tr&gt;</code></td>
</tr>
<tr>
<td>72</td>
<td><code>&lt;td colspan=&quot;3&quot;&gt;Estimated value&lt;/td&gt;</code></td>
</tr>
<tr>
<td>73</td>
<td><code>&lt;td align=&quot;right&quot;&gt;</code></td>
</tr>
<tr>
<td>74</td>
<td><code>&lt;!-- compute the estimated value of the cellar --&gt;</code></td>
</tr>
<tr>
<td>75</td>
<td><code>&lt;xsl:value-of select=&quot;sum (for $w in wine return $w/quantity * key('catalog',$w/@code)/cat:price)&quot;/&gt;</code></td>
</tr>
<tr>
<td>76</td>
<td><code>&lt;/td&gt;</code></td>
</tr>
<tr>
<td>77</td>
<td><code>&lt;td align=&quot;right&quot;&gt;</code></td>
</tr>
<tr>
<td>78</td>
<td><code>&lt;!-- compute the total number of bottles --&gt;</code></td>
</tr>
<tr>
<td>79</td>
<td><code>&lt;xsl:value-of select=&quot;sum(wine/quantity)&quot;/&gt;</code></td>
</tr>
<tr>
<td>80</td>
<td><code>&lt;/td&gt;</code></td>
</tr>
<tr>
<td>81</td>
<td><code>&lt;/tr&gt;</code></td>
</tr>
<tr>
<td>82</td>
<td><code>&lt;/table&gt;</code></td>
</tr>
<tr>
<td>83</td>
<td><code>&lt;!-- put comment section if at least one comment appears --&gt;</code></td>
</tr>
<tr>
<td>84</td>
<td><code>&lt;xsl:if test=&quot;count(wine/comment)&gt;0&quot;&gt;</code></td>
</tr>
<tr>
<td>85</td>
<td><code>&lt;h3&gt;Comments&lt;/h3&gt;</code></td>
</tr>
<tr>
<td>86</td>
<td><code>&lt;p&gt;</code></td>
</tr>
<tr>
<td>87</td>
<td><code>&lt;xsl:apply-templates select=&quot;wine/comment&quot;&gt;</code></td>
</tr>
<tr>
<td>88</td>
<td><code>&lt;xsl:sort select=&quot;../@code&quot; order=&quot;ascending&quot;/&gt;</code></td>
</tr>
<tr>
<td>89</td>
<td><code>&lt;/xsl:apply-templates&gt;</code></td>
</tr>
<tr>
<td>90</td>
<td><code>&lt;/p&gt;</code></td>
</tr>
<tr>
<td>91</td>
<td><code>&lt;/xsl:if&gt;</code></td>
</tr>
<tr>
<td>92</td>
<td><code>&lt;/xsl:template&gt;</code></td>
</tr>
<tr>
<td>93</td>
<td><code>&lt;!-- information about a wine --&gt;</code></td>
</tr>
<tr>
<td>94</td>
<td><code>&lt;xsl:template match=&quot;wine&quot;&gt;</code></td>
</tr>
<tr>
<td>95</td>
<td><code>&lt;tr&gt;</code></td>
</tr>
<tr>
<td>96</td>
<td><code>&lt;td&gt;&lt;xsl:value-of select=&quot;substring(@code,2)&quot;/&gt;&lt;/td&gt;</code></td>
</tr>
<tr>
<td>97</td>
<td><code>&lt;td&gt;</code></td>
</tr>
<tr>
<td>98</td>
<td><code>&lt;xsl:call-template name=&quot;nameAndUrl&quot;&gt;</code></td>
</tr>
<tr>
<td>99</td>
<td><code>&lt;xsl:with-param name=&quot;wine&quot; select=&quot;key('catalog',@code)&quot;/&gt;</code></td>
</tr>
<tr>
<td>100</td>
<td><code>&lt;/xsl:call-template&gt;</code></td>
</tr>
<tr>
<td>101</td>
<td><code>&lt;/td&gt;</code></td>
</tr>
<tr>
<td>102</td>
<td><code>&lt;td align=&quot;right&quot;&gt;&lt;xsl:value-of select=&quot;purchaseDate&quot;/&gt;</code></td>
</tr>
<tr>
<td>103</td>
<td><code>&lt;td align=&quot;center&quot;&gt;</code></td>
</tr>
<tr>
<td>104</td>
<td><code>&lt;xsl:value-of select=&quot;substring(****,1,&quot; if (rating/@stars) then rating/@stars else 0&quot;)/&gt;</code></td>
</tr>
<tr>
<td>105</td>
<td><code>&lt;/td&gt;</code></td>
</tr>
<tr>
<td>106</td>
<td><code>&lt;td align=&quot;right&quot;&gt;&lt;xsl:value-of select=&quot;quantity&quot;/&gt;</code></td>
</tr>
<tr>
<td>107</td>
<td><code>&lt;/td&gt;</code></td>
</tr>
<tr>
<td>108</td>
<td><code>&lt;/tr&gt;</code></td>
</tr>
<tr>
<td>109</td>
<td><code>&lt;/xsl:template&gt;</code></td>
</tr>
<tr>
<td>110</td>
<td><code>&lt;!-- output the name of the wine with a link for &quot;googling&quot; it --&gt;</code></td>
</tr>
<tr>
<td>111</td>
<td><code>&lt;xsl:template name=&quot;nameAndUrl&quot;&gt;</code></td>
</tr>
<tr>
<td>112</td>
<td><code>&lt;xsl:param name=&quot;wine&quot;/&gt;</code></td>
</tr>
<tr>
<td>113</td>
<td><code>&lt;xsl:variable name=&quot;name&quot; select=&quot;encode-for-uri($wine/@name)&quot;/&gt;</code></td>
</tr>
</tbody>
</table>
| 114 | `<!-- dynamic creation of an element and its attributes -->`
Parameter giving the start of the URL allowing a search for a given wine on Google.

Declaration of a key to access a wine element of the catalog use its code attribute as a key.

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).

Global header with the name of the owner and then a table giving more information about the owner and the location of the cellar.

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.

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.

Outputs the content of the cellar with the appropriate template.

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.

Applies the template for each wine but in increasing order of the code attribute.
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.

The total number of bottles is the sum of the values of all quantity elements.

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.

Outputs a line of an HTML table for a given wine.

Outputs the name of wine and its URL using the named template defined on line 113-4. The wine element in the catalog is found using the key function.

Outputs a string of * corresponding to the number of stars for this wine.

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.

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.

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.

A comment is preceded with the value of the surrounding code attribute and followed by a line break.

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>
```

A text node or an attribute is replaced by its string content.

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**

![HTML rendering of Example 5.6](image)

**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
<?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 supérieur" classification="a.c." code="C00043125" format="750ml"
<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>
</ul>
...</li>
</ul>
```

76
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 17-①) that produces the overall structure of the HTML file with its head and body elements. The processing of subelements (line 25-①) is called within an unnumbered list delimited by ul tags.
- matching attributes (line 31-①) 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 36-①) are transformed by outputting the name of the element returned by the function local-name() in bold (line 38-①) followed by its attributes. If the element is a node without any children (line 41-①) then it is output with only its content, otherwise (line 44-①) a new unnumbered list is started and template matching is applied on children nodes (line 46-①).
- text node content (line 53-①) is output within li tags.

The * in the match attribute (line 36-①) indicates that this rule applies to all element nodes not matched by a more specific rule such as the one on line 17-① 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)

```xml
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE stylesheet [ 
3     <!ENTITY space "&lt;xsl:text&gt; &lt;/xsl:text&gt;"> 
4 ]>
5 <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
6     xmlns="http://www.w3.org/1999/xhtml"
7     version="2.0">
8     <!-- to produce legal and validable XHTML ... -->
9     <xsl:strip-space elements="*"/>
10 <xsl:output method="xhtml"
11     doctype-public="-//W3C//DTD XHTML 1.0 Strict//EN"
12     doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
13     indent="yes"
14     encoding="UTF-8"/>
15 <xsl:template match="/">
16     <html>
17         <head>
18             <title>HTML compaction of"<xsl:value-of select="replace(document-uri(.),'\/*.\/*','$1')"/>
19         </head>
20         <body>
21             <ul>
22                 <xsl:apply-templates/>
23             </ul>
24         </body>
25     </html>
26 </xsl:template>
30 <xsl:template match="@*">
31     &space;<xsl:value-of select="local-name()"/>
32     <xsl:text>=""/><xsl:apply-templates select="."/>
33 </xsl:template>
35 <xsl:template match="*">
36     <li>
37         <b><xsl:value-of select="local-name()"/></b>
38             <xsl:apply-templates select="@*"/>
39             <xsl:choose>
40                 <xsl:when test="count(*)=0"> <!--single text node ?-->
41                     &space;"<xsl:value-of select="."/>
42                 </xsl:when>
43                 <xsl:otherwise> <!--possible mixed node-->
44                     <ul>
45                         <xsl:apply-templates/>
46                     </ul>
47                 </xsl:otherwise>
48             </xsl:choose>
49         </li>
50     </xsl:template>
```
Entity that corresponds to an explicit blank space.
Ignores blank spaces in all elements.
Matches the root node and produce the HTML skeleton.
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.
Start processing the root node of the document by applying the appropriate template.
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.
An element node is a list-item.
The name of the element in bold, followed by the attributes using the template defined on line 31.
The content of the element is output differently when it has children nodes or not.
There are no children nodes, in our case this will only be a text node, its content is copied in the current list item.
If there are children nodes, a new embedded unnumbered list is started.
The templates are applied on all children nodes, this is the default when no select attribute is present.
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.

```xml
<cellar-book [noNamespaceSchemaLocation="CellarBook.xsd"]>
  <wine-catalog [name="Prado Rey Roble"]>
    <appellation="Ribera-del-duero"/>
    <classification="d.o."/>
    <code="C00929026"/>
    <format="magnum"/>
    <properties [color="red"]>
      <alcoholic-strength="12.5"/>
      <nature="still"/>
    </properties>
    <origin [country="Spain"]>
      <region="Old Castille"/>
    </origin>
    <producer [value="Real Sitio de Ventosilla SA"]>
    </producer>
    <price [value="35.25"/>
    <year [value="2002"]>
  </wine>
  <owner [name="Jude"]>
    <family="Raisin"/>
    <street [value="1234 rue des Chateaux"]>
      <city [value="St-George"/>
    </city>
    <province [value="ON"]>
    </province>
    <postal-code [value="M7W 7S0"]>
    </postal-code>
  </owner>
  <location [street [value="4587 des Futailles"]>
    <city [value="Vallée des crus"]>
    </city>
    <province [value="QC"]>
    </province>
    <postal-code [value="H3C 4J8"]>
    </postal-code>
  </location>
  <cellar [code="C00043125"]>
    <purchaseDate [value="2005-06-20"]>
    </purchaseDate>
    <quantity [value="2"]>
    </quantity>
    <comment [value="Guy Lapalme, Montréal
       : should reorder soon"]>
    </comment>
  </cellar>
  ...
  <wine [code="C00929026"]>
    <purchaseDate [value="2003-10-15"]>
    </purchaseDate>
    <quantity [value="1"]>
    </quantity>
    <comment [value="for big
      parties"]>
    </comment>
  </wine>
</cellar-book>
```
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-\0. 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-\0 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-\0). 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-\0, we declare that all elements (*) of this stylesheet should ignore all spaces in the instance file.

On line 22-\0, 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-\0, 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 nl-and-indent (line 16-\0) 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

```xml
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="*"/>
   <xsl:output omit-xml-declaration="yes" method="text"/>

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

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

20 <xsl:template match="@*">
    <xsl:param name="indent"/>
    <xsl:if test="position()>1">
      <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:template>

25 <xsl:template match="text()">
    <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="normalize-space(.)"/>
</xsl:template>

30 <xsl:template match="*">
    <xsl:param name="indent"/>
    <xsl:if test="count(../@*)>0 or position()>1">
      <xsl:value-of select="gl:nl-and-indent($indent)"/>
    </xsl:if>
    <xsl:value-of select="local-name()"/>
    <xsl:apply-templates select="@*|*|text()">
      <xsl:with-param name="indent"
                      select="$indent + string-length(local-name())+1"/>
    </xsl:apply-templates>
</xsl:template>
```
The XSLT processor will ignore all whitespace nodes from the input so that the output only contains spaces explicitly inserted by the stylesheet.

Indicates that the output will be plain text, thus we do not want an XML declaration to be emitted.

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 \texttt{xsl:text} tag.

XPath function for outputting a number of spaces given by the \texttt{indent} parameter preceded by a new line indicated by a character entity.

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.

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.

Normalizes a text value, i.e. remove the surrounding spaces and line breaks.

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.

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
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

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 be 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) with two children:

- **fo:layout-master-set**, within the named template define-layout defined on line 32, describes the shape of a page with a fo:simple-page-master element (line 34) 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): here a simple repetition of our single page master.
- **fo:page-sequence** (line 58) refers to a page-sequence-master in which the content of the page will be given in the fo:flow element (line 60) 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) 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) and the traversal of the instance document starts on line 62 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-). 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 \texttt{fo:list-block} (created on line 60- of Example 5.10) is composed of \texttt{fo:list-items} one on top of each other. A \texttt{fo:list-item} (line 70-) is composed of a \texttt{fo:list-item-label} aligned horizontally with a \texttt{fo:list-item-body} (line 76-) 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 \texttt{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 \texttt{fo:label-end} which takes into account the value specified for the distance between blocks (line 61-). The starting position of the list item body must also be specified, most often again with a predefined function \texttt{fo:body-start} (line 76-).

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

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

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

Text nodes (line 116-) are output as a list item with an empty label i.e. an empty in-line block, again using the \texttt{labeledValue} named template, and the text content as body for the list item. Text nodes comprizing 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 \texttt{fo:marker} (line 87-) 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-).
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

```xml
<?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">
  <xsl:attribute-set name="page-size">
    <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>
  </xsl:attribute-set>

  <xsl:attribute-set name="block-decoration">
    <xsl:attribute name="border-color">black</xsl:attribute>
    <xsl:attribute name="border-left-style">solid</xsl:attribute>
    <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>
  </xsl:attribute-set>

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

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

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

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

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

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

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

70     <xsl:template match="@*">
     <xsl:call-template name="labeledValue">
     <xsl:with-param name="label">
     <xsl:value-of select="."/>
     </xsl:with-param name="label">
     </xsl:call-template>
</xsl:template>
<fo:inline xsl:use-attribute-sets="attribute-formatting">  
  @xsl:value-of select="local-name()"/>
</fo:inline>
</xsl:with-param>
<xsl:with-param name="value">
  <fo:inline font-style="normal" color="black">
    <xsl:value-of select="."/>
  </fo:inline>
</xsl:with-param>
</xsl:call-template>
</xsl:template>
<xsl:template match="text()">
  <xsl:variable name="content" select="normalize-space(.)"/>
  <xsl:if test="string-length($content)>0">
    <xsl:call-template name="labeledValue">
      <xsl:with-param name="label"/>
      <xsl:with-param name="value" select="$content"/>
    </xsl:call-template>
  </xsl:if>
</xsl:template>
<xsl:template name="labeledValue">
  <xsl:param name="label"/>
  <xsl:param name="value"/>
  <fo:list-item>
    <fo:list-item-label end-indent="label-end()">
      <fo:block>
        <xsl:copy-of select="$label"/>
      </fo:block>
    </fo:list-item-label>
    <fo:list-item-body start-indent="body-start()">
      <fo:block>
        <xsl:copy-of select="$value"/>
      </fo:block>
    </fo:list-item-body>
  </fo:list-item>
</xsl:template>
</xsl:stylesheet>

From the Instance Document to the XSL-FO file

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

Gets the current value of the marker defined on line 87.

Template that matches the root node and that defines the overall shape of the XSL-FO output enclosed in a fo:root.

Defines the layout by calling the template defined at line 32 followed by the sequence of pages.

The page sequence is filled out according to the definition of pages given on line 38. It first defines the content of the header by calling the named template on line 44 and then gives the content of the document in a single flow.

The content of the document to appear in the region-body is composed of a single list-block.

Specifies the distance between the start of its item and the start of its body.

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.

Template for an element.

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.

Outputs the element name with the formatting defined on line 22.

Outputs the content of an element depending on whether it has any children or attributes.

When there are no children nodes, the content of text nodes are copied.

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). Then starts a new fo:list-block to be filled by a recursive application of the templates on the content of all nodes.

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.

Applies the templates on all attributes, element and text nodes of this element.

An attribute is output as a one-element list using the named template defined on line 128. Note that the parameter values are complex XSL-FO elements.

The label parameter is the name of the attribute preceded by an @ with the appropriate formatting defined on line 27.

The value parameter is the value of the attribute (a text node).

A non-empty text node is output as a one-element list using the named template defined on line 128 with an empty label and the normalized text content as value parameter.

Named template for outputting a single list item with the content of two formal parameters.

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.

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 be 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 \texttt{class} or \texttt{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 is 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. [\texttt{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).
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 of Example 5.10.

All generated text that will appear before a tag will be in bold green.

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.

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.

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/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 2-6 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 &lt; entity because an XQuery query is not an XML expression.

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

- A more powerful type of query is a FLWOR expression\(^1\) (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.

```xml
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 FLOWER query bears some resemblance with the \texttt{for} loop in XPath 2.0. The \texttt{for} in this query (equivalent

\(^1\) pronounced \textit{flower} even though the letters are not in the proper order...
to line 11-14 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 14-17 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.

```xquery
<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.

```xquery
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

```xquery
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 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). Because the wine catalog is defined in a specific namespace, its prefix must be declared (line 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 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) 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) 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). 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.**

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

3 declare variable $color external;

5 (: to produce legal and validable XHTML ... :) :
   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-strict.dtd";
   declare option saxon:output "indent=yes";

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

declare function local:root-bis($catalog){
  element html {
    element head {element title {"Wine Catalog"}},
    element body {local:wine-catalog($catalog)}
  }
};

declare function local:wine-catalog($catalog){
  element h1 {concat("Wine Catalog (","$color"," only")")},
  element table {
    attribute border {1},
    element tr {
      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)
    }
  }
};

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

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

declare function local:toL($fmt){
  let $ml := local:toML($fmt)
  return
    if ($ml castable as xs:integer)
      then number($ml) div 1000
    else $ml
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-Ø) 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-Ø) 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-Ø creating a table with the address of the owner (line 28-Ø) and the cellar (line 31-Ø). It then calls the cellar function (line 37-Ø). The lines of the addresses are obtained by looping over all elements with a for and writing a <br/> 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-$$\bullet$$) with a position number greater than 1.

The cellar (line 40-$$\bullet$$) 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-$$\bullet$$), we can use the sum function. Finally, if there are any comments in the wine elements of the cellar (line 65-$$\bullet$$), 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-$$\bullet$$) to create a row in the table of wines. We first define a variable $code (line 76-$$\bullet$$) 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-$$\bullet$$). 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-$$\bullet$$) 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-$$\bullet$$). 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.dtd";
declare option saxon:output "indent=yes";

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

104
declare function local:cellar-book($cellar-book){
    <h1>Cellar of {data($cellar-book/owner/name)}</h1>,
    <table border="1">
        <tr>
            <th>Personal address</th>
            <th>Cellar address</th>
        </tr>
        <tr><td>{for $e in $cellar-book/owner/*[position()>1]
            return (data($e)<br/>)}
            <td>{for $e in $cellar-book/location/*
                return (data($e)<br/>)}
        </tr>
    </table>,
};

declare function local:cellar($cellar,$catalog){
    <table border="1">
        <tr>{
            for $t in ("Code","Name","Purchase Date","Rating","Nb Bottles")
                return <th>{$t}</th>}</tr>{
            for $w in $cellar/wine
                order by $w/@code ascending
                return local:wine($w,$catalog)}</tr>
    </table>,
    (: put comment section if at least one comment appears :) if (count($cellar/wine/comment)>0) then {


<h3>Comments</h3>

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

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

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

90 declare function local:comment($comment){
    data($comment/../@code),':',local:bold($comment),<br/>
};

95 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)}
        else $child
};

100 (: 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))}
        {QName($new-ns,local-name($element))}
        $element/@*,
    for $child in $element/node()
        return if ($child instance of element())
            then local:change-element-ns-deep($child,$new-ns)
        else $child
};

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

120
Parameter giving the start of the URL allowing a search for a given wine on Google.

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.

Global header with the name of the owner followed by a table giving more information about the owner and the location of the cellar.

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.

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.

Calls the function to output the content of the cellar.

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.

Calls the wine function for each wine but in increasing order of the code attribute.

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.

The total number of bottles is the sum of the values of all quantity elements.

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.

Outputs a line of an HTML table for a given wine.

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.

Outputs the name of the wine and its URL using the function defined on line 90.

Outputs a string of * corresponding to the number of stars for this wine.

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.

A comment is preceded with the value of the surrounding code attribute and followed by a line break.

A cat: bold element is transformed into an HTML b element.

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.

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 20-0) the function produces the overall structure of the HTML file with its head and body elements. The enclosed expression for processing subelements (line 27-0) is called within an unnumbered list delimited by ul tags.
- for an attribute (line 33-0), 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 37-0), the function produces the name of the element returned by the function local-name() in bold (line 39-0) 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 54-0) 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 options saxon:output "method=xml"; 
declare options saxon:output "doctype-public=-//W3C//DTD XHTML 1.0 Strict//EN"; 
declare options saxon:output "doctype-system=http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"; 
5 declare options saxon:output "indent=yes";

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

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

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

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

(: change the namespace of a subtree starting at an element
 adapted from Priscilla Walmsley, XQuery, O'Reilly, p. 258 :) ...

declare function local:change-element-ns-deep($element,$new-ns://${new-ns}{}
    element {QName($new-ns,local-name($element))} {
        $element/@*,
        for $child in $element/node()
            return if ($child instance of element())
            then local:change-element-ns-deep($child,$new-ns)
            else $child
    }
};

local:change-element-ns-deep( (: put the whole tree in the XHTML namespace :) ...

Declarations telling the SAXON transformation engine to serialize with the appropriate headers to
produce valid XHTML.
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.
Produces the XHTML skeleton for the document.
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.
Enclosed expression for starting the processing at the root element node.
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.
An element node is a *list-item*.

The name of the element in bold, followed by the attributes using the function defined on line 33-①.

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.

If there are children nodes, elements are processed recursively and text nodes (in the case of mixed content) are processed with the `text` function.

If there are no children nodes, it is then a text node that is copied to the output.

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.

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 29-① 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 21-①, 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 25-①, 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 5-①) 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 \texttt{nl-and-indent} (line 17-9) taking one parameter used to create a sequence of spaces, which are joined into one string preceded by a newline.

\textbf{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";
   declare option saxon:output "omit-xml-declaration=yes";

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

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

   declare function local:attribute($attribute){
      concat('@',local-name($attribute),'[',data($attribute),']')
   };
25 

   declare function local:text($text){
      normalize-space($text)
   };
30 

   declare function local:element($elem,$indent){
      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($newIndent)
                        else () ,
                     typeswitch ($child)
                        case attribute() return local:attribute($child)
                        case text() return local:text($child)
                        case element() return local:element($child,$newIndent)
                     default return ()
            )
         )
   }
```

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 [81]. For example, calling the query Example 6.1 and giving white as value of the $color parameter can be done as follows:

```java
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.

RDF [76] 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 metainformation 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 site: http://rali.iro.umontreal.ca
- a person: http://www.iro.umontreal.ca/lapalme
- an abstract concept: http://www.iro.umontreal.ca/lapalme/MyViewOnXML
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”!

115
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 [86]) 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 [77] which not only validates the input but can produce the corresponding graph or show the equivalent triples. Twinkle [87] 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

<table>
<thead>
<tr>
<th>Graphical form</th>
<th>Triple</th>
<th>Relational form</th>
<th>RDF/XML</th>
<th>Turtle</th>
</tr>
</thead>
</table>

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 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 and line 7. Another way of abbreviating URIs is to make them relative to another one using xml:base, see line 11 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 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.

```xml
1 <?xml version="1.0" encoding="UTF-8"?>
2     <!DOCTYPE RDF [     ①
3         <!ENTITY wc "http://www.iro.umontreal.ca/lapalme/WineCatalog#" >
4         <!ENTITY cb "http://www.iro.umontreal.ca/lapalme/CellarBook" >
5         ]>
6          xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" ③
7          xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
8          xmlns:wc="&wc;"
9          xmlns:cb="&cb;#"
10         xml:base="&cb;" ④
11     <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
12          xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
13          xmlns:wc="&wc;"
14          xmlns:cb="&cb;#">
15          <rdf:Description
16         rdf:about="http://www.iro.umontreal.ca/lapalme/CellarBook#theCellar" ⑤
17         <cb:ownedBy>
18         <!-- 1 -->
19         <rdf:Description rdf:about="&cb;#JudeRaisin"/>
20         </cb:ownedBy>
21         </rdf:Description>
22     </rdf:RDF> ⑥
23     <rdf:Description rdf:about="http://www.iro.umontreal.ca/lapalme/CellarBook#JudeRaisin"
24          cb:street="1234 rue des Châteaux" cb:city="St-George"
25          <cb:named rdf:parseType="Resource"> ⑥
26          <!-- 2 -->
27          <cb:first>Jude</cb:first>
28          <cb:family>Raisin</cb:family>
29         </cb:named>
30         </rdf:Description> ⑦
31     <rdf:Description rdf:about="#JudeRaisin"> <!-- 3 -->
32         <cb:loved rdf:resource="&wc;C00043125"/>
33         <cb:loved rdf:resource="&wc;C00871996"/>
34         <cb:comment rdf:parseType="Literal">this is <b>great</b>!</cb:comment>
35     </rdf:Description>
```
<rdf:Bag rdf:ID="theCellar">
<rdf:li rdf:resource="&wc;C00043125"/>
<rdf:li rdf:resource="&wc;C00042101"/>
<rdf:li rdf:resource="&wc;C00871996"/>
</rdf:Bag>

<rdf:Description rdf:about="#theCellar">
<cb:locatedAt rdf:nodeID="loc"/>
</rdf:Description>

<rdf:Description rdf:nodeID="loc">
<cb:street>4587 des Futailles</cb:street>
<cb:city>Vallée des crus</cb:city>
</rdf:Description>

<!-- Schema definition -->
<!-- Classes -->
<rdfs:Class rdf:ID="Person"/>
<rdfs:Class rdf:ID="Cellar"/>
<rdfs:Class rdf:ID="Wine"/>

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

<rdf:Property rdf:ID="ownedBy">
<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 [76]. 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 3-9, 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 35-9. 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.
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 29-5). The predicate is indicated by an element named by its URI using the namespace prefix.

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.

#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.

The cellar-book, referred by the id CellarBook (corresponding to the full URI given in line 13-9) 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.

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 45-8. Contrarily to a node referred by a rdf:ID, the id of a blank node, referred by rdf:nodeId is local to this file.

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.

Start of the RDFS part that will be further explained in the next section.

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.

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 54-8. 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).

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 [86]. 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 < and >; an URI can also be written as a QName i.e. an identifier preceded by a namespace prefix and a colon. A namespace prefix is defined with a @prefix
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.

A 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.

A 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

<table>
<thead>
<tr>
<th>Blank Node</th>
<th>Corresponding Triples</th>
</tr>
</thead>
<tbody>
<tr>
<td>[] ex:p &quot;o&quot; .</td>
<td>__:b0 ex:p &quot;o&quot; .</td>
</tr>
<tr>
<td>[ex:p &quot;o&quot;] .</td>
<td>__:b1 ex:p &quot;o&quot; .</td>
</tr>
<tr>
<td>[ex:p &quot;o&quot;] ex:q ex:o2 .</td>
<td>__:b2 ex:p &quot;o&quot; .</td>
</tr>
<tr>
<td>[ex:p ex:o;ex:p1 &quot;ol&quot;]</td>
<td>__:b4 ex:p ex:o .</td>
</tr>
</tbody>
</table>

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

Jena [68] 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 . t t1] 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 22-9. 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#> . ①
@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 ③
  rdf:_1 wc:C00043125 ;
```

①
③
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.

Six predicates sharing cb:CellarBook as subject.

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.

Four predicates having cb:JudeRaisin as subject are grouped, their predicate and object being separated by semicolons.

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.

Two objects in namespaces wc also share both their subject cb:JudeRaisin and their predicate cb:loved, so they are separated by commas.

Start of the RDFS part further described in the next section.

Definition of three classes in cb namespace. A class declaration is represented in RDF by a 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.

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 24-6. A property definition also adds a a link between the name of the property and the predefined rdf:Property.
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- of Example 7.1 and line 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 constrast 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 explicitely in the file. For example, from line 29- of Example 7.1, or the equivalent in line 18- of Example 7.2, where we declare that the property `cb:loved` links a person with a wine line 59- of Example 7.1 (line 28- 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- of Example 7.1 (line 28- 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 ressource be a member of many classes even ones that do not make sense for a human, e.g. a ressource 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 SPARQL\(^1\), 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 \textit{variable}, noted by an identifier preceded by \texttt{?} or \texttt{\$}. Such a triple is called \textit{triple pattern}, many triple patterns can be combined to form a \textit{graph pattern}. For example, in Example 7.2: the triple pattern

\[
\text{cb:JudeRaisin cb:street } ?\text{address}
\]

matches the triple

\[
\text{cb:JudeRaisin cb:street "1234 rue des Châteaux"}
\]

The triple

\[
\text{cb:JudeRaisin cb:loved } ?\text{wine}
\]

matches two triples corresponding to the two wines.

\[
?\text{s } ?\text{p } ?\text{o}
\]

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,

\[
\text{cb:CellarBook cb:ownedBy } ?\text{owner.}
\]

\[
?\text{owner cb:loved } ?\text{wine.}
\]

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

\[
\text{cb:CellarBook cb:ownedBy [cb:loved } ?\text{wine].}
\]

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

\[
\text{SELECT vars where } \{ \text{graph pattern} \}
\]

\[
\text{CONSTRUCT } \{ \text{graph pattern}_1 \} \text{ vars where } \{ \text{graph pattern}_2 \}
\]

The output of this query is a table containing the values of the selected variables from all triples that match the graph pattern. 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.

\(^1\)SPARQL 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#>  ①
  PREFIX cb: <http://www.iro.umontreal.ca/lapalme/CellarBook#>
  PREFIX wc: <http://www.iro.umontreal.ca/lapalme/WineCatalog#>

5 # show all triples
   SELECT * WHERE {?s ?p ?o}  ②

   # show the loved wines of the CellarBook
   SELECT * WHERE {[cb:loved ?wine]}  ③

10 # show the street on which the owner of the Cellarbook lives
   SELECT ?street WHERE {cb:CellarBook cb:ownedBy [cb:street ?street]}  ④

   # show all wines of the cellar
   SELECT ?wine WHERE {cb:CellarBook ?x ?wine FILTER regex(str(?x),"_[0-9]+$")}  ⑤

   # define a new triple with an inverse relation

① Define namespace prefixes to be used in triples. The prefix declaration has slightly different syntax than the one used in Turtle (see line 1-① of Example 7.2)
② Show all triples. * indicates that the value of all variables used in the queries will be displayed.
③ 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.
④ Use the bracket notation to simplify getting information out of a complex graph structure.
⑤ 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.
⑥ Create a new triple from an existing triple.

Twinkle [87] 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 [82]. 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;
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 26-1) creates a DocumentBuilderFactory object (line 34-1) 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 36-1). To validate against an XML Schema another one must be set (line 38-1). 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 41-1) by using a factory method which returns a DocumentBuilder object to which an error handler (line 5-1 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 83-1) is called (line 46-1) on the root element.

In order to simplify further processing, we define (line 66-1) 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 76-1) that are not interesting for further processing; some care has to be taken to save the next sibling (line 71-1) 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 78-1) to strip-space from the current child node. When it is called on the document element (line 45-1), it will go through all the document to remove empty text nodes and nodes that are not text or elements.

Within compact (line 83-1), the processing depends on the type of element obtained on line 85-1. If it is an element node (line 86-1), we first print the node name followed by an opening bracket. On line 91-1, 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 97-1, is a simple traversal algorithm with a recursive call to compact (line 100-1), followed by the printing of a closing bracket. In the case of a text node (line 106-1), 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

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

---

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

128
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.FactoryConfigurationError;
import javax.xml.parsers.ParserConfigurationException;
import org.xml.sax.InputSource;
import org.xml.sax.ErrorHandler;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
import java.io.IOException;
import javax.swing.JTree;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.tree.DefaultMutableTreeNode;
import javax.swing.tree.DefaultTreeModel;

public class DOMCompact{

    public static void main(String argv[]){
        // is there anything to do?
        if (argv.length != 1) {
            System.out.println("Usage: java DOMCompact file");
            System.exit(1);
        }
        // parse file
        try {
            DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
            factory.setValidating(true);
            factory.setNamespaceAware(true);
            DocumentBuilder builder = factory.newDocumentBuilder();
            builder.setErrorHandler(new CompactErrorHandler());
            Document doc = builder.parse(argv[0]);
            Node docElem=doc.getDocumentElement();
            stripSpace(docElem);
            compact(docElem,"");
            System.out.println();
            new TreeViewer(
                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());
        } catch (SAXException e) {
            System.out.println(e.getMessage());
        } catch (ParserConfigurationException e) {
            System.out.println("Parser configuration error");
        } catch (IOException e) {
            System.out.println("IO Error on "+argv[0]);
        }
    }
}

public class CompactErrorHandler implements ErrorHandler{
    public CompactErrorHandler(){
        // do nothing
    }
    // override the method
    public void error(SAXParseException e){
        System.out.println(e.getMessage());
    }
}

public class DOMCompact{
    public static void main(String argv[]){
        // is there anything to do?
        if (argv.length != 1) {
            System.out.println("Usage: java DOMCompact file");
            System.exit(1);
        }
        // parse file
        try {
            DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
            factory.setValidating(true);
            factory.setNamespaceAware(true);
            DocumentBuilder builder = factory.newDocumentBuilder();
            builder.setErrorHandler(new CompactErrorHandler());
            Document doc = builder.parse(argv[0]);
            Node docElem=doc.getDocumentElement();
            stripSpace(docElem);
            compact(docElem,"");
            System.out.println();
            new TreeViewer(
                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());
        } catch (SAXException e) {
            System.out.println(e.getMessage());
        } catch (ParserConfigurationException e) {
            System.out.println("Parser configuration error");
        } catch (IOException e) {
            System.out.println("IO Error on "+argv[0]);
        }
    }
}
private static void stripSpace(Node node) {
    Node child = node.getFirstChild();
    while (child != null) {
        // save the sibling of the node that will
        // perhaps be removed and set to null
        Node c = child.getNextSibling();
        if ((child.getNodeType() == Node.TEXT_NODE &&
             child.getNodeValue().trim().length() == 0) ||
            (child.getNodeType() != Node.TEXT_NODE &&
             (child.getNodeType() != Node.ELEMENT_NODE)))
            node.removeChild(child);
        else // process children recursively
            stripSpace(child);
        child = c;
    }
}

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

Beginning of the Java program that checks if there is a parameter to be used as a file name to compact.
Beginning of the parsing process creating a document builder that will be used for validation and for creating the DOM structure of the document.
Indicates that document validation will be performed, on top of the default well-formedness check.
Validation should be done with a Schema, not a DTD.
Creates a class that will be used for creating the DOM structure.
Parses the document using the previously created objects.
Saves a reference to the root document element.
Remove all white space nodes and non text or element nodes from the whole DOM structure.
Creates a compact form by traversing the DOM structure from the root. compact is defined on line 83-
Creates an interactive window representing the DOM structure.
Deals with exceptions that can occur during the parsing phase, such as non well-formedness, validation error or file-access exceptions.
Empty text children nodes are removed so that only meaningful content nodes are compacted.
Saves a reference to the next sibling.
Removes the current child from the DOM structure.
Strips space recursively on this child node.
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.
Determines the type of the current node in order to choose the appropriate processing step.
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 that the name of the current node.
Attributes are printed on separate (indented) lines; an attribute name is prefixed with an @ and the value is enclosed in square brackets.
The children are processed with the current indentation. Finally, a closing bracket is added.
Compact this child recursively.
Normalize and prints a text node.
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

```java
import org.xml.sax.ErrorHandler;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;

public class CompactErrorHandler implements ErrorHandler{
    private void message(String mess, SAXParseException e) throws SAXException{
        System.out.println("\n"+mess+
```
Implementation of XML parsing error handling.

Prints out an error message with the current error position. This method is used for all three kinds of errors.

An irrecoverable error.

An XML validation error.

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 29-①) and sets flags for validation. The parser is obtained on line 32-① and a property is set to indicate that we want validation to be done with an XML Schema. Parsing is then started on line 36-① 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

```java
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
import org.xml.sax.helpers.XMLReaderFactory;

import org.xml.sax.helpers.DefaultHandler;
import javax.xml.parsers.SAXParserFactory;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.parsers.SAXParser;
import java.io.IOException;
import javax.swing.JTree;
import javax.swing.JFrame;
import javax.swing.JScrollPane;

public class SAXCompact {
    private static JTree jtree = new JTree();

    public static void main(String argv[]) {
        if (argv.length != 1) {
            System.out.println("Usage: java SAXCompact file");
            return;
        }

        // XMLParser creation
        SAXParserFactory factory;
        SAXParser saxParser;
        try {
            factory = SAXParserFactory.newInstance();
            factory.setNamespaceAware(true);
            factory.setValidating(true);
            saxParser = factory.newSAXParser();
            saxParser.setProperty(
                              "http://java.sun.com/xml/jaxp/properties/schemaLanguage",
                                "http://www.w3.org/2001/XMLSchema",
                                "http://www.w3.org/2001/XMLSchema"");

            // parse file and print compact form
            saxParser.parse(argv[0], new CompactHandler());
            System.out.println();

            // parse file and build a tree form
            saxParser.parse(argv[0], new JTreeHandler(jtree));
            // display the built tree
            new TreeViewer(jtree).show();
        } catch (ParserConfigurationException e) {
            System.out.println("Bad parser configuration");
        } catch (SAXParseException e) {
            System.out.println(argv[0] + " is not well-formed");
            System.out.println(e.getMessage() +
                " at line "+e.getLineNumber()+
                ", column "+e.getColumnNumber());
        } catch (SAXException e) {
            System.out.println(argv[0] + " is not well-formed");
            System.out.println(e.getMessage() +
                " at line "+e.getLineNumber()+
                ", column "+e.getColumnNumber());
        }
    }
}
```
Initialises an empty JTree for interactive output.

Main method that first checks if a file name has been given as an argument to the program.

Creates a document factory.

Creates a new SAX parser.

Parses the file given as argument while sending the parse events to a handler that will output the compact form.

Parses the file given as argument while sending the parse events to a handler that will add nodes to a JTree for interactive viewing.

Displays the interactive tree viewer.

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 20-1), endElement (line 38-9) and characters (line 45-9) 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 66-9 is called. When this happens, we call the message method (line 58-9) 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

```java
import org.xml.sax.SAXParseException;
import org.xml.sax.helpers.DefaultHandler;
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;

public class CompactHandler extends DefaultHandler {

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

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

    private boolean closed = false;  // closed mode?
    protected int indent;           // current indentation value

    // ...

    public void characters(char[] ch, int start, int length) {
        // Write character data to JTree
    }

    public void endElement(String uri, String localName, String qName) throws SAXException {
        // End element event
    }

    public void startElement(String uri, String localName, String qName, Attributes attrs) throws SAXException {
        // Start element event
    }

    // ...

    // Handle exceptions that can occur in both parsing and input file processing
}
```
public void startElement(String uri, String localName,
        String raw, Attributes attrs)
    throws SAXException {
    if (closed) {
        System.out.print('\n' + blanks(indent));
        closed = false;
    }
    indent = indent + 1 + localName.length();
    System.out.print(localName + '[');
    // deal with attributes
    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;
    }
}

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

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

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

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

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

1. 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.
2. 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.
3. 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.
4. Method used to output error messages.
5. 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 file\(^2\). 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-\(2\)). 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-\(3\)) 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-\(3\)) with the current state of the XML stream which will process the current element and its children before returning.

Within `compact` (line 47-\(4\)), the type of current token is checked and if it is the starting tag of an element (line 49-\(5\)) then the name of the element is printed and the indentation is updated. Then attributes and values are output properly indented (line 53-\(6\)). Each non empty children is then printed by a recursive call to

\(^2\)There 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-Φ). A closing bracket is output when the end element tag is encountered which terminates the loop. The character content (line 68-Φ) 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

```java
import java.io.FileInputStream;
import java.io.IOException;
import javax.swing.JTree;
import javax.swing.tree.DefaultTreeModel;
import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamException;
import javax.xml.stream.XMLStreamReader;

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

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

    private static void compact(XMLStreamReader xmlsr, String indent) {
        // ...
    }
}
```

[Stream API for XML (StAX)](http://www.renderx.com/)
Main method that first checks if a file name has been given as an argument to the program.

Creates an input factory.

Creates a new stream parser.

Ignores the tokens that come before the first element (e.g. processing instructions or DTD).

Calls the compacting process with the current token and then prints a newline to flush the content of the last line.

Releases the data structure of the last streaming operation before opening the file for the second time, to build the JTree.

Handles exceptions that can occur in both parsing and input file processing.

Method to compact from the current token.

If it is a start element tag, outputs the name of the element followed by an opening bracket and update the current indentation.

Outputs each attribute name and value all indented except the first one.

Loops on children nodes that are not whitespace and compacts each of them with the correct indentation.

Recursive call to the compacting process.

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.
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) 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) 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) is a static method so it is called with the instruction TreeViewer.jTreeBuild( ). (line 21 and line 48 of Example 8.2). It uses the same algorithm as compact (line 83 of Example 8.1) by recursively processing element or text DOM nodes. In the case of an element node (line 25), 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) 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**

```java
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;

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

public class TreeViewer extends JFrame{
    TreeViewer(JTree jtree) {
        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) {
    if (node == null)
        return null;
    switch (node.getNodeType()) {
25         case Node.ELEMENT_NODE: {
            DefaultMutableTreeNode treeNode =
            new DefaultMutableTreeNode(node.getNodeName());
            NamedNodeMap attrs = node.getAttributes();
            for (int i = 0; i < attrs.getLength(); i++)
                treeNode.add(
                new DefaultMutableTreeNode(
                    '@'+attrs.item(i).getNodeName()+
                    '['+attrs.item(i).getNodeValue()+'']);
            // process children
30            Node child = node.getFirstChild();
            while(child != null){
                DefaultMutableTreeNode childTree = jTreeBuild(child);
                if(childTree!=null)
                    treeNode.add(childTree);
                child = child.getNextSibling();
            }
            return treeNode;
        }
    case Node.TEXT_NODE: {
45        String text = node.getNodeValue().trim();
        return text.length()==0 ? null
            : new DefaultMutableTreeNode(text);
    }
    }
}
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at [http://www.renderx.com/](http://www.renderx.com/)
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 13-①). When an XML element start-tag is encountered, in startElement (line 19-①), a new tree node is created and made the current node (line 25-①). Its attributes are then added as children (line 29-①). A text node is simply added as a child of the current node (line 41-①). When an end-tag is encountered (line 34-①), we set the current node to the parent of the current node. The constructor (line 15-①) only keeps a copy of the JTree that will be displayed. Its model is initialized on the first call of startElement (line 22-①).

To create a JTree, we add (line 40-① 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 42-①) of the TreeViewer class (line 13-① of Example 8.6). The JTree instance is built by calling the SAX parser and giving it a JTreeHandler (line 40-①).

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 36-① of Example 8.3) and once more for the tree display (line 40-①).

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

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

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

public class JTreeHandler extends DefaultHandler{

    private DefaultTreeModel treeModel;
    private DefaultMutableTreeNode node = null;
    private JTree jtree;

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

    public void startElement(String uri, String localName,
        Attributes attrs) throws SAXException {
        super.startElement(uri,localName,raw,attrs);
        if(node==null) //initialise node and model
            jtree.setModel(
                new DefaultTreeModel(
                    node=new DefaultMutableTreeNode(localName)));
        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++)
            node.add(new DefaultMutableTreeNode("@"+attrs.getLocalName(i)+"["+attrs.getValue(i)+"]");
    }
```
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-ⓝ) 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-⛭ of Example 8.1) by recursively processing element or text DOM nodes. In the case of an element node (line 55-⛭), it creates a new DefaultMutableTreeNode for the element and adds attributes as its first children (line 65-⛭); it then processes each child by recursively building its subtree (line 65-⛭) which is added as a child of the current node. In the case of character content (line 72-⛭), a single node with this character content is returned.

8.4.3. Building a JTree with StAX

The JTree node currently being built.

1. Initializes the JTree structure to which the JTree nodes will be added.
2. Deals with a start element.
3. Calls the start method of the superclass.
4. If the node is not initialized, creates a new tree model and uses it for the root.
5. Creates a new node and adds it to the current JTree node. Note that the new current node is updated to this new node.
6. Adds attributes as nodes of the current node.
7. Processes the end tag.
8. Calls the endElement method of the superclass and the current node is now the parent of the current node.
9. Processes a text node by creating a new JTree node with text as its content if it is not empty.
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 [70] 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 [75], Python [40], Haskell [50], Prolog [53], Ruby [78], 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 that 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:

- `builder.newDocument()` creates an empty document to which elements can be added
- `doc.createElement(String s)` creates a new element named `s` in document `doc`
- `parent.appendChild(Element e)` adds element `e` as the last child of element `parent`; if `e` was already the child of another node, it is removed before being assigned to its new parent
- `e.setAttribute(String name, String value)` adds the attribute `name` with the corresponding `value` to the element `e`; if the attribute already exists, its value is replaced
- `doc.createTextNode(String s)` creates a new text node with content `s` in document `doc`

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-*) first creates a `Document` instance (line 29-*) which will hold the XML tree. It then creates a specialized tokenizer (line 30-*) from the file name given as argument to the program. It goes on to find the name of the root element (line 34-*) and creates the root node. It then calls the `expand` method (line 39-*) 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-*) and use the document as source and `System.out` as output (line 46-*). We also set an `output property` so that the output is nicely indented (line 43-*).

Expansion (line 52-*) is a recursive process that adds element and text nodes to a parent node received as a parameter; it processes each attribute (line 55-*) by getting the name and value of the attribute and adding it to the current element line 58-*; the content of the element (line 62-*) 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-*); otherwise, a text node is created (line 68-*)

---

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at [http://www.renderx.com/](http://www.renderx.com/)
Example 9.1. [CompactTokenizer.java]: Specialized stream tokenizer that ignores blank tokens

```java
import java.io.Reader;
import java.io.IOException;
import java.io.StreamTokenizer;

public class CompactTokenizer {
    private StreamTokenizer st;

    CompactTokenizer(Reader r) {
        st = new StreamTokenizer(r);
        st.resetSyntax(); // remove parsing of numbers...
        st.wordChars(' ', 'ÿ'); // everything is part of a word except the following...
        st.ordinaryChar('
');
        st.ordinaryChar('[');
        st.ordinaryChar(']');
        st.ordinaryChar('@');
    }

    public String nextToken() throws IOException {
        st.nextToken();
        while (st.ttype == '
' ||
               (st.ttype == StreamTokenizer.TT_WORD &&
                st.sval.trim().length() == 0))
            st.nextToken();
        return getToken();
    }

    public String getToken() {
        return (st.ttype == StreamTokenizer.TT_WORD) ? st.sval : ("" + (char) st.ttype);
    }

    public String skip(String sym) throws IOException {
        if (getToken().equals(sym))
            return nextToken();
        else
            throw new IllegalArgumentException("skip: "+sym+" expected but"+sym +" found ");
    }
}
```

1 Constructor that receives a `Reader` and creates a customized `StreamTokenizer`.
2 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.
3 Calls the Java tokenizer and skips newlines and empty text nodes. It returns the current token as a `String`.
4 Function for accessing the current token as a `String`. 
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.

```java
import org.w3c.dom.Element;
import org.w3c.dom.Document;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.transform.OutputKeys;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.TransformerConfigurationException;
import org.w3c.dom.DocumentBuilder;
import org.w3c.dom.Document;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.transform.OutputKeys;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.TransformerConfigurationException;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import java.io.IOException;
import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.InputStreamReader;
import java.io.StreamTokenizer;

public class DOMExpand {
    //
    public static void main(String[] argv) {
        try {
            DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
            DocumentBuilder builder = factory.newDocumentBuilder();
            Document doc = builder.newDocument();
            CompactTokenizer st = new CompactTokenizer(
                new BufferedReader(
                    new InputStreamReader(
                        new FileInputStream(argv[0]))));
            String rootName = "dummyElement";
            while(!st.nextToken().equals("["))
                rootName=st.getToken();
            expand((Element)doc.appendChild(doc.createElement(rootNam));
            TransformerFactory tFactory = TransformerFactory.newInstance();
            Transformer transformer = tFactory.newTransformer();
            transformer.setOutputProperty(OutputKeys.INDENT,"yes");
            DOMSource source = new DOMSource(doc);
        }
    }
}
```

Creating a DOM Document

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/
StreamResult result = new StreamResult(System.out);
transformer.transform(source,result);
} catch ( Exception ex ) {
ex.printStackTrace();
}
}

static void expand(Element elem) throws IOException{
    Document doc = elem.getOwnerDocument();
st.skip("[");
    while(st.getToken().equals("@")){ // process attributes
        String attName = st.nextToken();
        st.nextToken();
        elem.setAttribute(attName,st.skip("[")));
        st.nextToken();
st.skip("]");
    }
    while(!st.getToken().equals("]")){ // process element
        String s = st.nextToken().trim();
        st.nextToken();
        if(st.getToken().equals("["))
            expand((Element)elem.appendChild(doc.createElement(s)));
        else{
            elem.appendChild(doc.createTextNode(s));
        }
st.skip("]");
}
}

1. Main procedure for expanding a compact form.
2. Creates a new empty DOM document.
3. Creates a special purpose tokenizer, described in Example 9.1, to process the file whose name is given as a parameter of the program.
4. 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.
5. Expands the rest of the file as a child of the root just created.
6. Creates an identity transformer for serializing the output.
7. Makes the serializer pretty-print the output.
8. Creates a transformation source from the DOM structure created.
9. 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.
10. Finds a reference to the document node necessary for creating new element and text nodes.
11. Loops on all attribute name and value pairs.
12. Adds the attribute to the current element.
13. Loops on all elements until the closing bracket.
14. 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.
15. 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 20-0); 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 25-0).

Example 9.3. [SAXExpand.java]: XML document creation using SAX events

```java
import org.xml.sax.InputSource;
import javax.xml.transform.sax.SAXSource;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.TransformerConfigurationException;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
public class SAXExpand {
    public static void main( String[] argv ) {
        try {
            InputSource inputSource =
                new InputSource(
                    new BufferedReader(new FileReader(argv[0])));
            CompactReader saxReader = new CompactReader();
            SAXSource source = new SAXSource(saxReader,inputSource);
            StreamResult result = new StreamResult(System.out);
            TransformerFactory tFactory = TransformerFactory.newInstance();
            Transformer transformer = tFactory.newTransformer();
            transformer.transform(source,result);
        } catch (TransformerException ex){
            System.out.println("TransformerException"+ex);
            ex.printStackTrace();
        } catch (IOException ex) {
            System.out.println("IOException"+ex);
            ex.printStackTrace();
        }
    }
}
```
First, a Reader is created to be used as an `InputSource`.

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 is 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 89-91).

The SAX parsing events are sent to an event handler (line 18-19) for which we define (get... and set...) accessor methods available to the user of the handler. The main method is `parse` (line 41-42) that uses the same algorithm as the ones used for the DOM approach. `parse` uses a `CompactTokenizer` instance (line 39-40) 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 400-401) and then calls `expand` (line 50-51) 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 64-65). Once all the attributes have been gathered, we can send a `startElement` event to the handler (line 72-73). We then process the content of the event either with a recursive call to `expand` (line 77-78) or by creating a text element by sending a `characters` event to the handler (line 80-81). Once all children elements have been processed, an `endElement` event is sent to the handler (line 85-86).

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 71-72) or end element (line 84-85) or before a new text node (line 79-80).

Example 9.4. [CompactReader.java]: Compact form parsing to generate SAX events

```java
import org.xml.sax.XMLReader;
import org.xml.sax.ContentHandler;
import org.xml.sax.DTDHandler;
import org.xml.sax.EntityResolver;
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;
import java.io.IOException;
import java.io.StreamTokenizer;
import java.util.Arrays;

public class CompactReader implements XMLReader{
    private String nsu = ""; // no namespace URI
    private ContentHandler handler;
    
    
```
private static char[] blanks = "\n    ".toCharArray();

private void ignorableSpacing(int nb) throws SAXException {
    if (nb > blanks.length) {// extend the length of space array
        blanks = new char[nb];
        blanks[0] = '\n';
        Arrays.fill(blanks, 1, blanks.length, ' ');
    }
    handler.ignorableWhitespace(blanks, 0, nb);
}

public ContentHandler getContentHandler() {return handler;}

public void setContentHandler(ContentHandler handler) {
    this.handler = handler;
}

private CompactTokenizer st;

public void parse(InputSource input) {
    try{
        String rootName = "dummyRoot";
        st = new CompactTokenizer(input.getCharacterStream());
        while (!st.nextToken().equals("[")) {
            rootName = st.getToken();
        }
        handler.startDocument();
        expand(rootName, 1);
        ignorableSpacing(1);
        handler.endDocument();
    } catch (SAXException e) {
        System.out.println(e.getMessage());
    } catch (IOException e) {
        System.out.println("IO Error:" + e);
    }
}

void expand(String elementName, int indent)
    throws IOException, SAXException {
    AttributesImpl attrs = new AttributesImpl();
    st.skip("[");
    while (st.nextToken().equals("@")) {// process attributes
        String attName = st.getToken();
        st.nextToken();
        attrs.addAttribute(nsu, attName, attName, "CDATA", st.skip("[");
        st.nextToken();
        st.skip("]");

    }
    ignorableSpacing(indent);
    handler.startElement(nsu, elementName, elementName, attrs);
while(!st.getToken().equals("]")){ // process content
    String s = st.getToken().trim();
    st.nextToken();
    if(st.getToken().equals("["))
        expand(s,indent+3);
    else {
        ignorableSpacing(indent+3);
        handler.characters(s.toCharArray(),0,s.length());
    }
}
    st.skip("]");
    ignorableSpacing(indent);
    handler.endElement(nsu,elementName,elementName);
}

// dummy definitions...
public DTDHandler getDTDHandler(){return null;}
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){}
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){}
}

① Private variable for keeping track of the document handler.
② Character array containing a carriage return followed by a certain number of blanks that will used for formatting.
③ Method for returning blanks that will be ignored by the parser.
④ Variable for the tokenizer that handles the compact output.
⑤ Starting point of the parsing process. Creates the tokenizer for the input file and generates a startDocument event.
⑥ Finds the name of the root node.
⑦ Starts the recursive processing of the content of an element with the name of the root node.
⑧ Recursive processing of an element content.
⑨ Loops over the content of all attributes that are inserted into a single AttributesImpl structure.
⑩ Generate spacing for pretty-printing the output structure.
⑪ Generates a startElement event with the name of the element and the structure containing the attributes.
⑫ Loops over all children nodes in order to process either element or text nodes.
⑬ If it is an element node, calls expand recursively but with more indentation.
⑭ If it is a text node, generates a characters event with an appropriate indentation
⑮ Indentation before a text node.
⑯ Indentation before closing an element
⑰ Indicates the end of an element.
⑱ 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:

- `xmlsw.writeStartDocument();` initialises an empty document to which elements can be added.
- `xmlsw.writeStartElement(String s)` creates a new element named s.
- `xmlsw.writeAttribute(String name, String value)` adds the attribute name with the corresponding value to the last element produced by a call to writeStartElement. It is possible to add attributes as long as no call to writeStartElement, writeCharacters or writeEndElement has been done.
- `xmlsw.writeEndElement` closes the last started element.
- `xmlsw.writeCharacters(String s)` creates a new text node with content s as content of the last started element.

Like we did in Section 9.1, to simplify parsing, we create a customized StreamTokenizer (line 18-()) 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-()) 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-()) and call expand method (line 29-()) 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-()) 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-()) first outputs the attributes on the current element (line 46-()) and then writes either a new element and calls itself recursively to process its contents (line 57-()) or it outputs the character content of the element (line 63-()).

Example 9.5. [StAXExpand.java]: XML document creation using the StAX approach

```java
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;

public class StAXExpand {
    static XMLStreamWriter xmlsw = null;
    public static void main(String[] argv) {
        try {
            xmlsw = XMLOutputFactory.newInstance().createXMLStreamWriter(System.out);
            CompactTokenizer tok = new CompactTokenizer(new FileReader(argv[0]));

            String rootName = "dummyRoot";
            // ignore everything preceding the word before the first "["
            while(!tok.nextToken().equals("["){
                rootName=tok.getToken();
            }

            // start creating new document
            xmlsw.writeStartDocument();
            ignorableSpacing(0);
            xmlsw.writeStartElement(rootName);
            expand(tok,3);
            ignorableSpacing(0);
            xmlsw.writeEndElement();
            xmlsw.flush();
            xmlsw.close();
        } catch (XMLStreamException e){
            System.out.println(e.getMessage());
        } catch (IOException ex) {
            System.out.println("IOException"+ex);
            ex.printStackTrace();
        }
    }

    public static void expand(CompactTokenizer tok, int indent) throws IOException,XMLStreamException {
        tok.skip("[");
        while(tok.getToken().equals("") ){ // add attributes
            String attName = tok.nextToken();
            tok.nextToken();
            xmlsw.writeAttribute(attName,tok.skip("[");
            tok.nextToken();
            tok.skip("]");
        }

        boolean lastWasElement=true; // for controlling the output of newlines
        while(!tok.nextToken().equals("]")){ // process content
            String s = tok.getToken().trim();
            tok.nextToken();
            if(tok.getToken().equals("["){
                if(lastWasElement)ignorableSpacing(indent);
                xmlsw.writeStartElement(s);
                expand(tok,indent+3);
                lastWasElement=true;
            }
        }
    }
}

Document Creation by Programming in Java

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/
} else {
    xmlsw.writeCharacters(s);
    lastWasElement=false;
}

} else {
    tok.skip("\]");
    if(lastWasElement)ignorableSpacing(indent-3);
    xmlsw.writeEndElement();
}

private static char[] blanks = "\n".toCharArray();
private static void ignorableSpacing(int nb)
    throws XMLStreamException {
    if(nb>blanks.length){// extend the length of space array
        blanks = new char[nb+1];
        blanks[0]="\n";
        Arrays.fill(blanks,1,blanks.length,' ');
    }
    xmlsw.writeCharacters(blanks, 0, nb+1);
}

1. Creates the XMLStreamWriter for outputting the XML text.
2. Creates a specialized tokenizer for the compact form.
3. Initializes the XML document.
4. Calls the expanding process on the root element.
5. Terminates the XML writing process by ending the document and closing the file.
6. Deals with exceptions that can be raised during writing the XML code and the file.
7. Recursive method to output an element properly indented.
8. Adds attributes name and value on the last element outputted.
9. Deals with the content of the element.
10. Recursively calls the expansion on the content of the element.
11. Outputs the character content.
12. Outputs the end of the current element.
13. 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 [75], Python [40], Haskell [50], Prolog [53], Ruby [78] 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 [71] 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 xmlint 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 [72].

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, and PHP [74], a server-side script language for the Internet. We will then describe EcmaScript for XML (E4X)[66], which has been recently standardized. EcmaScript is the standard for JavaScript and ActionScript, the programming language for Flash. Flash 9 implements the latest EcmaScript specifications and thus allows the E4X XML notation within its programs. With the growing interest in AJAX technologies that mix Javascript and XML, it can be expected that other Javascript implementations will also allow this XML notation within any normal Javascript program. We will finally describe to process XML with Swift [83], a statically typed script language.

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) [69] and YAML (YAML Ain't a Markup Language) [88]. 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 [79] 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'  
include REXML

def compact(node,indent)
  if(node.class==Element)
    print node.name+"["
    indent += ' '*(node.name.length+1);
    first=true;
    node.attributes.each do |key,value|
      print "\n#{indent}" unless first
      print "@#{key}[#{value}]"
      first=false;
    end
    node.children.each do |child|
      if child.class==Element || child.value.strip.length>0
        print "\n#{indent}" unless first
        compact(child,indent)
        first=false;
      end
    end
    print "]"
  elsif node.class==Text
    print node.value.strip.gsub(/ *\n */,"\n") # normalize new lines
  end
end
Makes sure that the Document class in the REXML library is present. Includes all classes of the REXML module.

Prints the content of the XML node. Each line prefixed with an indentation, a string composed of the appropriate number of spaces.

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.

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.

Processes all children with an iterator.

If it is an element or a non-empty text node, recursively calls compact possibly changing line if it not the first child.

Processes a text node by printing it and normalizing internal newlines.

Parses the file by creating a new XML document from the content of the standard input.

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.

```ruby
1 require 'rexml/document'
require 'rexml/parsers/sax2parser'
require 'rexml/sax2listener'
require 'CompactHandler'

5 parser = REXML::Parsers::SAX2Parser.new(STDIN)
parser.listen(CompactHandler.new)
parsed.parse
```

Makes sure that the appropriate SAX parser classes of the REXML library and the CompactHandler class are present.

Creates an instance of the SAX parser that will parse the standard input.

Has the parser send its parsing events to the CompactHandler class described in Example 10.3.

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.

```ruby
1 class CompactHandler
    include REXML::SAX2Listener
    def initialize
        @closed=false
      @indent=0;
      @entities = {"&"=>"&", "&quot;=>"", "&apos;"=>"", "&lt;" => "<", "&gt;" => ">"}
    end
    def start_element(uri,localname,qname,attributes)
        if @closed
            print "\n"+" "*@indent
        @closed=false
        end
      @indent += 1+localname.length()
      print localname+"[
      first=true;
      attributes.each do |key,value|
          print "\n"+" "*@indent if !first
      first=false
      print "@#{key}[#{expandEntities(value)}]"
      @closed=true
      end
    end
    def end_element(uri,localname,qname)
        print "]
        @closed=true
      @indent=@indent-1-localname.length
    end
    def characters(text)
        if text.strip.length>0
            if @closed
                print "\n"+" "*@indent
            @closed=false
            end
        text.strip!                # remove leading and trailing space
        text.gsub!(/\s\r\n\]+/,' ')    # normalize space
      print expandEntities(text)
      @closed=true
      end
    end
    def entitydecl(name,decl)
        @entities["&"+name+";"]=decl
      end
end
```

Alternative approaches

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/
def expandEntities(text)
    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
            if res=text.match("&#x([0-9a-fA-F]+);") # hexadecimal
                text=res.pre_match+[res[1].hex].pack("U")+res.post_match
            elseif res=text.match("&#([0-9]+);")     # decimal
                text=res.pre_match+[res[1].to_i].pack("U")+res.post_match
            else
                break
            end
        end
    end
end

Includes the default declarations needed to handle events sent by a SAX parser.
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.
When a new element is started, finishes the current indentation if needed.
Updates the current indentation by adding the length of the element name.
Prints the first attribute on the same line and the others on the subsequent lines properly indented.
When an element finishes, closes the current bracket and updates the current indentation.
For a non-empty text node, ends current line if needed.
Removes leading and trailing space, normalizes and expands entities.
When a new entity is declared, add it to the entities table.
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.
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.
includes the StringScanner package on which this tokenizer is built.

Defines the CompactTokenizer class.

Makes the current token available as a readable attribute.

@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.

Gets the next token but skip those containing only whitespace. It returns the value of the current token.

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 an 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.

```ruby
1 require 'rexml/document'
   include REXML

2 require 'compactTokenizer'
5 st = CompactTokenizer.new(STDIN)

# start new document
6 doc = Document.new()
7 doc << XMLDecl.new(1.0,"UTF-8")
10 # find the name of the root
15 rootName = 'dummyElement'
16 while st.nextToken!='['
18   rootName=st.token
20 end

15 # create the element and return it with the current token
20 def expand(st,elementName)
23   elem = Element.new(elementName)
24   st.skip('[')
28   while st.token=='@' # process attributes
29     attName = st.skip('@')
30     st.nextToken
31     elem.attributes[attName] = st.skip('[')
33   st.nextToken
35   st.skip(']')
39 end
40   while st.nextToken!=']' # process children
41     s = st.token.strip
42     st.nextToken
43     if(st.token=='[')
44       elem << expand(st,s)
45     else
46       elem << Text.new(s)
47     end
51 st.skip(']')
52 return elem
54 end

40 # expand from the root element
45 doc << expand(st,rootName)
46 # write it properly indented
49 doc.write(STDOUT,0)
```

1 Ensures that the library REXML document class is loaded, then include it.
2 Ensures that the CompactTokenizer (Example 10.4) is loaded.
Alternative approaches

① Creates the tokenizer on the standard input.
② Initializes a new XML document.
③ Adds an appropriate XML declaration.
④ The name of the root is the first token immediately followed by an opening square bracket.
⑤ Creates an XML element corresponding to the content of rootName and returns it.
⑥ Creates a new, empty XML element with name elementName.
⑦ 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.
⑧ Adding an attribute is done by assigning to the attributes hash within the XML node.
⑨ 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.
⑩ A child node is expanded by a recursive call whose result is added as the last child of the current element.
⑪ A text node is added as the last child of the current element.
⑫ Calls the expansion of the document starting from the rootName and adds the result as the child of the document.
⑬ 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:
             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
        for i in range(len(attrs)):
            if not first: sys.stdout.write('\n'+indent)
```
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.
SAX parsing using Python

1 import xml.sax
   import sys
   import CompactHandler

5 parser = xml.sax.make_parser()
p2.parser.setContentHandler(CompactHandler.CompactHandler())
p3.parser.parse(sys.stdin)
p4.sys.stdout.write(\n"

10

1 Imports the SAX parser package and the CompactHandler class.
2 Creates an instance of the SAX parser.
3 Makes the parser send its parsing events to the CompactHandler class described in Example 10.8.
4 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
   import sys

   class CompactHandler(ContentHandler):
5     def __init__(self):
         self.closed = False
         self.indent = 0
         self.lastNodeWasText = False

10     def startElement(self, localname, attrs):
         if self.closed:
             sys.stdout.write(\n"+self.indent*" )
             self.closed = False
             self.indent+=1+len(localname)
         sys.stdout.write(localname+']
         first = True
         for name in attrs.getNames():
             if not first:
                 sys.stdout.write(\n"+self.indent*" )
             first = False
             sys.stdout.write('@'+name+'["'+attrs.getValue(name)+'"]')
             self.closed = True
             self.lastNodeWasText = False

25     def endElement(self, localname):
         self.closed = True
         sys.stdout.write(']')

167
30 def characters(self, data):
    data = data.strip()
    if(len(data)>0):
        if self.closed and not self.lastNodeWasText:
            sys.stdout.write('
'+self.indent*' ')
            self.closed = False
            sys.stdout.write(data)
            self.closed=True
        self.lastNodeWasText = True

① Imports the xml.sax.handler package containing the ContentHandler class which will be subclassed to handle events sent by a SAX parser.
② 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.
③ When a new element is started, finishes the current indentation if needed.
④ Updates the current indentation by adding the length of the element name.
⑤ Prints the first attribute on the same line and the others on the subsequent lines properly indented.
⑥ When an element finishes, closes the current bracket and updates the current indentation.
⑦ 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

class XMLStreamReader():
    ""
    XMLStreamReader: simplified interface to DOMEventStream so that it looks similar to the Java XMLStreamReader with similar methods
    It concatenates all successive text nodes
    ""
    def __init__(self, file):
        self.events = pulldom.parse(file)
        self.event = None
        self.node = None
        self.look_ahead=None

① Alternative approaches
XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/
# 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
Imports the pulldom package.

1. Defines a new class to simplify access to the pull parsing process.
2. 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.
3. Gets the next event either from stream if there is no look-ahead or from the look-ahead.
4. Concatenates successive character nodes. The first non-text node is kept in the look-ahead for the next call.
5. A series of node type tests on the current event.
6. Checks if the current element is a start node, otherwise raises an exception. Useful for debugging the following methods.
7. Returns information about the current start element.
8. 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.

```python
from XMLStreamReader import XMLStreamReader
import sys

def compact(xmlsr,indent):
    if xmlsr.isStartElement():
        sys.stdout.write(xmlsr.getLocalName() + '[')
        indent += (len(xmlsr.getLocalName()) + 1) * ' '
        count = xmlsr.getAttributeCount()
        for i in range(count):
            if i > 0:
                sys.stdout.write(indent)
            sys.stdout.write('@' + xmlsr.getAttributeLocalName(i) + '[' + xmlsr.getAttributeValue(i) + ']
            first = count == 0
    while True:
        xmlsr.next()
        while xmlsr.isWhiteSpace(): xmlsr.next()
        if xmlsr.isEndElement(): break;
        if first:
            first = False
        else:
            sys.stdout.write(indent)
            compact(xmlsr, indent);
        sys.stdout.write(']')
    elif xmlsr.isCharacters():
        sys.stdout.write(xmlsr.getText().strip())

xmlsr = XMLStreamReader(sys.stdin)
xmlsr.next()
while not xmlsr.isStartElement(): xmlsr.next()
```

Alternative approaches

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at [http://www.renderx.com/](http://www.renderx.com/)
Imports the XMLStreamReader class defined in Example 10.9
If it is a start element tag, outputs the name of the element followed by an opening bracket and update the current indentation.
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.
Loops on children nodes that are not whitespace and compacts each of them with the correct indentation.
Recursive call to the compacting process.
Prints the normalized character content.
Creates a new stream reader, indicates that it will parse the standard input.
Ignores the tokens that come before the first element (e.g. processing instructions).
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 a 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.

```python
1 import re, sys

2 class CompactTokenizer():

5     def __init__(self,file):
         self.pat = re.compile('([\[\@\]])')
         self.whitespacepat = re.compile('
? *$')
         self.file = file
         self.tokens = []

10     def nextToken(self):
         while True:
             if len(self.tokens)==0:
                 self.line = self.file.readline().strip()
             else:
                 if self.line:
                     self.tokens = [tok for tok in
                     self.pat.split(self.line) if tok!='']
             else:
```
Includes the regular expression \texttt{re} package on which this tokenizer is built.

Defines the \texttt{CompactTokenizer} class.

Defines the \texttt{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.

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 \texttt{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.

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.

Returns the current token.

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 \texttt{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 \texttt{appendChild} method that adds the new node as the last child of a given node. The attributes of a node are added using the \texttt{setAttribute} method.

\textbf{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.
ct.nextToken()
10         elem.setAttribute(attName,ct.skip("[")))  
ct.nextToken()
ct.skip("]")
while ct.getToken()!=']':
    s=ct.getToken().strip()
15         ct.nextToken()
if ct.getToken()=="[":
    expand(elem.appendChild(doc.createElement(s.strip())))
else:
    elem.appendChild(doc.createTextNode(s))
20     ct.skip("]")
return elem

doc = xml.dom.minidom.Document()
ct = CompactTokenizer(sys.stdin)
25 while ct.nextToken()!='[':
    rootName=ct.getToken()
    expand(doc.appendChild(doc.createElement(rootName.strip())))

print (doc.toprettyxml("  "))

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
Alternative approaches
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 as ElementTree object and the traversing it with standard
Python iterators to create a compact version of the file.
1 import xml.etree.ElementTree as ET
import sys, re

5

10

15

20

25

def stripNS(tag):
return re.sub("^\{.+\}","",tag)
def compact(node,indent):
if node==None:return
if ET.iselement(node):
localname = stripNS(node.tag)
sys.stdout.write(localname+'[')
indent += (len(localname)+1)*" "
attrs = node.attrib
first=True
for name in attrs:
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:
sys.stdout.write(node.text.strip())
first=False
for child in list(node):
if not first: sys.stdout.write(indent)
compact(child,indent)
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")
sys.stdout.write('\n')

Imports the ElementTree class and renames is as ET because it will be used often. Imports other
necessary packages.

174

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/


As ElementTree nodes contain namespace information in their print names, we discard it here to keep only the local name.

Function to compact a node (first parameter) with each new line preceded by an indentation given by the second parameter.

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.

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.

If there is text content, prints it. The text is normalized by removing spaces at the start and end.

If the element has children, compacts them, a new line is output if it is not the first child.

Calls compact recursively.

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.

```python
1 import xml.etree.ElementTree as ET
  import sys,re
  from CompactTokenizer import CompactTokenizer

5 def expand(elem):
    global ct
    ct.skip('[')
    while ct.getToken()=='@':
        attName = ct.nextToken()
        ct.nextToken()
        elem.attrib[attName]=ct.skip('[')
        ct.nextToken()
        ct.skip(']')
    lastNodeWasText=True
    while ct.getToken()!=']':
        s=ct.getToken().strip()
        ct.nextToken()
        if ct.getToken()==['):
            child=ET.Element(s)
            elem.append(child)
            expand(child)
        else:
            if lastNodeWasText:
                if not elem.text:
                    elem.text=""
```

Other means of dealing with XML documents using Python
Imports the ElementTree, other system packages and the CompactTokenizer (Example 10.11). 

Adds the content of the file corresponding to the children of the current node to the elem element.

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.

Adding an attribute is done by setting the attribute value associated with its name in the attrib dictionary of the current element.

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.

A new element named s is created and its children are filled in by a recursive call to expand.

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.

Creates the tokenizer on the standard input.

The name of the root is the first token immediately followed by an opening square bracket.

Creates the root node as an ElementTree.

The root node which is filled in by a call to expand.

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.
 Outputs the name of element followed by its attributes on a single line.

 If there is a text, outputs it.

 Recursively process each children possibly writing the text content of the `tail` property at the same level of indentation. Finally closes the tag.

 Prints the document node on the standard output by calling the `pprint` method defined above.

 Other means of dealing with XML documents using Python
10.3. XML processing with PHP

PHP [74] 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 accommodated. It features also many libraries for reading, creating and transforming XML data [75]. 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.

```php
1  <?php
2   function compact_($node,$indent){
3       if ($node->nodeType==XML_ELEMENT_NODE) {
4           print $node->nodeName."[";
5           $indent.=str_repeat(' ',strlen($node->nodeName)+1);
6           $first=true;
7           foreach ($node->attributes as $attrName => $attrValue) {
8               if(!$first)print "\n".$indent;"@".$attrName."[".$attrValue->value."]";
9               $first=false;
10           }
11           foreach ($node->childNodes as $child)
12               if($child->nodeType==XML_ELEMENT_NODE or
13                  strlen($child->textContent)>0 ){
14                   if(!$first)print "\n".$indent;"[
15                   compact_($child,$indent);
16                   $first=false;
17               }
18           print "]";
19       } else if ($node->nodeType==XML_TEXT_NODE)
20           print preg_replace("/ *\n? +/", " ",trim($node->textContent));
21   }
22
23   $dom = new DomDocument();
24   $dom->preserveWhiteSpace = false;
25   $dom->substituteEntities = true;
26
```
$dom->load("php://stdin");
compact_($dom->documentElement,"");
print "\n";
30 ?>

1 Prints the content of the XML node. Each line prefixed with an indentation, a string composed of the
2 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
4 appropriate number of spaces to the current indentation.
5 Deals with attributes which are contained in an array over which we iterate. For each attribute, prints
6 a @, its key and the corresponding value within square brackets. A new line is started for each attribute
7 except the first.
8 Processes all children with a for-each loop.
9 If it is an element or a non-empty text node, recursively calls compact possibly changing line if it
10 not the first child.
11 Processes a text node by printing it and normalizing internal newlines.
12 Parses the file by creating a new XML document from the content of the standard input. Sets parameters
13 to that white space only nodes are not returned and that entities are replaced by their values before the
14 XML processing.
15 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  function tag($tag,$body,$attrs="") {
    return "\<$tag".".$attrs">$body</$tag";
  }
5  function compact_($node){
    if ($node->nodeType==XML_ELEMENT_NODE) {
      $res="";
      $attrs="";
10     foreach ($node->attributes as $key => $value)
        $attrs.=" $key='".$value'";
      $res.=tag("b",".$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){
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.

Returns a string corresponding to the content of the XML node.

If the node is an element, the name of the node is and its attributes are embedded in a \texttt{b} tag. The content of the attributes and the children nodes is embedded in a \texttt{li} tag.

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 \texttt{b} tag.

Gets the list of all children.

If there are more than one child, accumulates the content of the children and wrap it within an \texttt{ul} tag.

In the case of one child, returns its content.

Processes a text node by normalizing its content.

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.

Calls the compacting process on the document node within an HTML structure forming the shape of HTML page. \texttt{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 \texttt{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 \textit{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.

```php
1 <?php
   require 'CompactHandler.php';
   $handler = new CompactHandler();

5 $parser = xml_parser_create("UTF-8");
   xml_parser_set_option($parser, XML_OPTION_TARGET_ENCODING, "UTF-8");  
   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);
   xml_set_element_handler($parser, "startElement", "endElement");
   xml_set_character_data_handler($parser,"characters");

15 $fp = fopen("php://stdin","r");
18 while ($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)));
   }
   print "\n";
   xml_parser_free($parser);
?>
```

1 Makes sure that the appropriate CompactHandler class is present.
2 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.
3 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.
4 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.
Alternative approaches

```php
<?php

class CompactHandler {
    var $closed, $indent, $chars;

    function CompactHandler(){
        $this->closed=false;
        $this->indent=0;
        $this->chars="";
    }

    function startElement($parser, $localname, array $attributes){
        $this->flushChars();
        if ($this->closed) {
            print "\n".str_repeat(' ', $this->indent);
            $this->closed=false;
        }
        $this->indent+=1+strlen($localname);
        print $localname."[";
        $first=true;
        foreach ($attributes as $attrName => $attrValue){
            if(!$first)print "\n".str_repeat(' ', $this->indent);
            print "@".$attrName."[".$attrValue."]";
            $first=false;
            $this->closed=true;
        }
    }

    function endElement($parser, $localname){
        $this->flushChars();
        print "]";
        $this->closed=true;
        $this->indent-=1+strlen($localname);
    }

    function characters($parser, $text){
        if(strlen(trim($text))>0){
            if ($this->closed) {
                print "\n".str_repeat(' ', $this->indent);
                $this->closed=false;
            }
            $this->chars=$this->chars.trim($text);
        }
    }

    function flushChars(){
        if (strlen($this->chars)>0) {
            if ($this->closed) {
                print "\n".str_repeat(' ', $this->indent);
                $this->closed=false;
            }
            print preg_replace("/ \n? \+/", "\n", $this->chars);
            $this->closed=true;
            $this->chars="";
        }
    }

    ...
}
```

Alternative approaches
XML to PDF by RenderX XEP XSL-FO Formatter, visit us at [http://www.renderx.com/](http://www.renderx.com/)
Declares variables to save the state of the parsing processing between calls to parsing call-back methods.

Initializing two instance variables needed to output the element with the correct indentation and one to accumulate the character content of successive character nodes.

When a new element is started, outputs the content of last character nodes and finishes the current indentation if needed.

Updates the current indentation by adding the length of the element name.

Prints the first attribute on the same line and the others on the subsequent lines properly indented.

When an element finishes, outputs the contents of the last character nodes and closes the current bracket and updates the current indentation.

For a non-empty text node, ends current line if needed and adds its contents to the $chars string.

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

```php
function compact_($xmlsr,$indent){
    if($xmlsr->nodeType==XMLReader::ELEMENT){
        print $xmlsr->name."["
        $indent.=str_repeat(' ',strlen($xmlsr->name)+1);
        $first=true;
        if($xmlsr->hasAttributes){
            while($xmlsr->moveToNextAttribute()){  
                if(!$first)print 
$indent="@".$xmlsr->name."[".$xmlsr->value."]
                $first=false;
            }
        } // Warning: PHP does not return an END_ELEMENT on a empty tag
        if(!$xmlsr->isEmptyElement)
            while(true){
                do {($xmlsr->read());
                    while ($xmlsr->nodeType==XMLReader::SIGNIFICANT_WHITESPACE);
                    if($xmlsr->nodeType==XMLReader::END_ELEMENT)break;
                    if($first)$first=false;
                    else print "$\nindent";
                }compact_($xmlsr,$indent);
            }
}
```
Alternative approaches

```php
}$xmlsr->nodeType==XMLReader::TEXT)
25       print preg_replace("/ */
? +/", " ", trim($xmlsr->value));
else {
   die("STRANGE NODE:

26   "$xmlsr->nodeType."\n"));
}

30
$xmlsr = new XMLReader();
$xmlsr->open("php://stdin");
$xmlsr->setParserProperty(XMLReader::SUBST_ENTITIES,true);
while ($xmlsr->nodeType!=XMLReader::ELEMENT)$xmlsr->read();
compact_($xmlsr,""));
35   compact_($xmlsr,""));
   print "\n";
?>
```

1. 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.
2. If it is a start element tag, outputs the name of the element followed by an opening bracket and update the current indentation.
3. Outputs each attribute name and value all indented except the first one. Attributes are obtained by iteration using the the moveToNextAttribute method.
4. 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.
5. Recursive call to the compacting process.
6. Prints the normalized character content.
7. Creates a new stream parser, indicates that it will parse the standard input and that entities must be substituted by the parse.
8. Ignores the tokens that come before the first element (e.g. processing instructions).
9. 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.
<?php

class CompactTokenizer {
    var $index, $tokens;

    function CompactTokenizer($file){
        global $index,$tokens;
        $tokens = preg_split("/
|(\[|\]|@)/",
            file_get_contents($file),
            -1,PREG_SPLIT_NO_EMPTY|PREG_SPLIT_DELIM_CAPTURE);
        $index = 0;
    }

    function getToken(){
        global $tokens,$index;
        $token = $index<count($tokens)?$tokens[$index]:false;
        return $token;
    }

    function nextToken(){
        global $tokens,$index;
        do {
            $index=$index+1;
            $token = $this->getToken();
            if($token==false)break;
            $token=trim($token);
        } while (strlen($token)==0);
        return $token;
    }

    function skip($sym){
        if($this->getToken()==$sym)
            return $this->nextToken();
        die("skip:$sym expected but ".$this->getToken().' found. index='.$index');
    }
}

\n
defines the \CompactTokenizer class with two instance variables $tokens an array to keep all tokens and $index to indicate the current token.

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.

Check that the index of the current token is within the bounds of the array and set it to false otherwise. Return it.

Gets the next token but skip those containing only whitespace. It returns the value of the current token.

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.

```php
1 <?php
   require 'CompactTokenizer.php';

   function expand($elem) {
5     global $st, $dom;
     $st->skip('[');
     while ($st->getToken() == '@') {  // process attributes
       $attName = $st->skip('@');
       $st->nextToken();
       $elem->setAttribute($attName, $st->skip('['));
       $st->nextToken();
       $st->skip(']');
     }
     while ($st->getToken() != false && $st->getToken() != ']') {
15       $s = trim($st->getToken());  // process children
       $st->nextToken();
       if ($st->getToken() == ']')
         expand($elem->appendChild($dom->createElement($s)));
       else
         $elem->appendChild($dom->createTextNode($s));
     }
     $st->skip(']');
   }

25 $st = new CompactTokenizer('php://stdin');
   $dom = new DOMDocument();
   $rootname = $st->getToken();
   while ($st->nextToken() != '[')
     $rootname = $st->getToken();
   expand($dom->appendChild($dom->createElement($rootname)));

   $dom->formatOutput = true;
   print $dom->saveXML();
   ?>
```

Ensures that the CompactTokenizer (Example 10.20) is loaded.

Adds the content of the file corresponding to the children of the current node to the `$elem` element.

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.

Adding an attribute is done by calling the `setAttribute` method of the current element.
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.

A new element named \( s \) is created and its children are filled in by a recursive call to `expand`.

A text node is added as the last child of the current element.

Creates the tokenizer on the standard input.

Initializes a new XML document.

The name of the root is the first token immediately followed by an opening square bracket.

Creates the root node which is filled in by a call to `expand`.

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.

```php
1 <?php
2     $xml = new DOMDocument();
3     $xml->load("php://stdin");

5     $xsl = new DOMDocument();
6     $xsl->load('compact-php.xsl');

7     $proc = new XSLTProcessor();
8     $proc->importStyleSheet($xsl);
10    $proc -> setParameter(null, 'name','value'); // unused in this example

11    print $proc->transformToXML($xml);
?>
```

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 `SimpleXML`Expressions 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.

```php
<?php
    $cellarbook = simplexml_load_file("CellarBook.xml");

    p("/cellar-book/owner", $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]);
    p("//postal-code/..", $cellarbook->xpath("//postal-code/.."));
    p("/cellar-book/owner/street", $cellarbook->owner->street);
    foreach ($cellarbook->cellar->wine as $w) {
        p("//wine/@code", $w["code"]);
    }
    foreach ($cellarbook->children("http://www.iro.umontreal.ca/lapalme/wine-catalog")->children() as $w) {p("//wine/@code", $w["code"]);
}
    p("/cellar-book/cellar/wine[1]/comment", $cellarbook->cellar->wine[0]->comment);
    $sum=0;
    foreach($cellarbook->cellar->wine as $w){
        $sum+=(int)$w->quantity;
    }
    p("sum(/cellar-book/cellar/wine/quantity)",$sum);

    print("for \$w in //wine return concat(\\$w/quantity,':',//cat:wine/@code[.=\\$w/@code]/../@name)"");
```
Other means of dealing with XML documents using PHP

```php
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("--------------
";
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&amp;lt;20]",
$cheapFrenchWines);

50 function p($xpath,$sxo){
    print("$xpath\n");
    var_dump($sxo);
    print("--------------\n");
}
55 ?>
```

1. Loads the file given as parameter and returns the root element.
2. 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.
3. 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.
Alternative approaches

```php
<?php
function compact_($sxelem,$indent){
    print $sxelem->getName()."[";
    $indent.=str_repeat(' ',strlen($sxelem->getName())+1);
    $first=true;
    foreach ($sxelem->attributes() as $attname => $attvalue) {
        if(!$first)print "$indent";
        print "@{$attname}[$attvalue]";
        $first=false;
    }
    $text = trim($sxelem);
    if(strlen($text)){
        print preg_replace("/ *
? +/"," ",$text);
        $first=false;
    }
    $children = $sxelem->children();
    foreach ($children as $child){
        if(!$first)print "$indent";
        compact_($child,$indent);
        $first=false;
    }
    print "]";
}

$dom = simplexml_load_file('php://stdin');
compact_($dom,"");
print "\n";
?>
```

1 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_.
2 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.
3 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.
4 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.
5 if the element has children, compacts them, a new line is output if it is not the first child.
6 Calls compact_recursively.
7 Reads an XML instance file, here the standard input and keeps a reference on the root node.
8 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. [SimpleXMLExpand.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.

```php
<?php
require 'CompactTokenizer.php';

function expand($elem) {
    global $st;
    $st->skip('[');
    while ($st->getToken() == '@') { // process attributes
        $attName = $st->skip('"');
        $st->nextToken();
        $elem->addAttribute("$attName", $st->skip(DateTime::T)));
    $st->nextToken();
    $st->skip(']');
    }
    while ($st->getToken() != false && $st->getToken() != ']') {
        $s = trim($st->getToken()); // process children
        $st->nextToken();
        if ($st->getToken() == '[') {
            expand($elem->addChild($s));
        } else {
            $elem[0] = $s; // add a text node
        }
    $st->skip(']');
}

$st = new CompactTokenizer('php://stdin');
$rootname = $st->getToken();
while ($st->nextToken() != '[') {
    $rootname = $st->getToken();
}

$sxelem = new SimpleXMLElement("<$rootname/>");
expand($sxelem);

print ppXML($sxelem, "");

function ppXML($sxelem, $indent) {
    $res = "";
    $name = $sxelem->getName();
    $res .= "$indent<$name";
    if (count($sxelem->attributes()) > 0) {
        foreach ($sxelem->attributes() as $attname => $attvalue) {
            $res .= "$indent $attname=" . $attvalue . "";
        }
    }
    $text = trim((string) $sxelem);
    if ($sxelem->count() == 0 && strlen($text) == 0) {
        $res .= "/>
    } else if ($sxelem->count() == 0) {
        $res .= "$text</$name>
    } else {
        $res .= "$text<br/>
    }
}
```

Other means of dealing with XML documents using PHP
else {
    $res.=">$text\n";
    foreach ($sxelem->children() as $child) {
        $res.=ppXML($child,$indent."   ");
        $res.="$indent</$name>\n";
    }
    return $res;
}
?>

1. Ensures that the CompactTokenizer (Example 10.20) is loaded.
2. Adds the content of the file corresponding to the children of the current node to the $elem element.
3. 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.
4. 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 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.
8. Creates the tokenizer on the standard input.
9. The name of the root is the first token immediately followed by an opening square bracket.
10. Creates the root node as a SimpleXMLElement.
11. The root node which is filled in by a call to expand.
12. 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.
13. 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.
14. Outputs the name of element followed by its attributes on a single line.
15. If there is no text content and no children nodes, closes the current tag.
16. If there are not children node, outputs the text and closes the tag.
17. 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 E4X

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:

```javascript
var wine:XML = <wine name="Château La Piquette" format="1l">
    <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.

```javascript
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 10.1 gives a few examples of E4X expressions for accessing information in the cellar book.
Figure 10.1. 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
   cellarBook.cellar.wine.(quantity<=2)
2 cellarBook.cellar.wine[0]
3 cellarBook.descendants("*").(elements("postal-code").length()>0)
4 5 cellarBook.owner.street
   cellarBook..wine.@code
5 var catNS:Namespace
   = new Namespace("cat","http://www.iro.umontreal.ca/lapalme/wine-catalog");
   cellarBook.addNamespace(catNS);
10 cellarBook..catNS::wine.@code

var wines:XMLList = cellarBook..wine;
   wines[wines.length()-1].@code
15 cellarBook.cellar.wine[0].comment.catNS::bold

var sum:Number=0;
   for each (var q in cellarBook.cellar.wine.quantity)
   sum += Number(q);
20 sum

var res:String="";
   for each (var w in cellarBook..wine)
   res+=w.quantity+":"+cellarBook..catNS::wine.(@code==w.@code).@name+"\n";
25 res
```

*1* The owner element of the cellar. Result: node on line 103.

*2* 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.

*3* The first wine of the cellar. Result: node on line 120.

*4* 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.

*5* The street of the cellar's owner. Result: "1234 rue des Châteaux".

*6* 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".

*7* 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".

*8* 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".

194
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;).

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.

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 Montaguèret, 1:Prado Rey Roble.

Sequence of French wines in the catalog costing less than 20 dollars. Result: wines that start on lines 24 and 42.

### 10.4.1. 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 10.26 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 10.26. [DOMCompact.as]** Text compaction of the cellar book (Example 2.2) with E4X using the DOM model

Compare this program with Example 8.1.

```actionscript
1 package{
   public class DOMCompact{
      var doc:XML;

5         function DOMCompact(s:String){
            doc = XML(s);
        }

10         public function toString():String {
            return compact(doc);
        }

15         var blanks:String = "     ";
      private function spaces(n:int){
            while(blanks.length<n)blanks+="     ";
            return blanks.substr(0,n);
        }

20         private function compact(node:XML,indent:String=" "):String {
            var type:String = node.nodeKind();
            switch (type) {
               case "element":
                  var out:String = node.localName()+"[
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/
The class with one instance variable, the document of type XML.

Constructor of the class that creates an XML document from a string. Parsing is done through the XML function.

Produces a compact string by calling the compact function.

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.

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.

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.

All attributes are output with their names and values within brackets.

Each child node is output with a recursive call to compact.

A text node is the content of the node but normalized by replacing newlines and their surrounding whitespaces with a single space.

10.4.2. 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 10.27. [CompactTokenizer.as] E4X specialized string scanner that returns tokens of compact form

Compare this program with Example 9.1.
package {
    class CompactTokenizer {
        private var tokens:Array = [];
        private var tok:String = "";

        function CompactTokenizer(source:String):void{
            tokens=source.split(/(\[|\]|\n|\@|[^\[\]\n@])/).filter(notEmpty);
        }

        private function notEmpty(element:*,index:int,a:Array):Boolean{
            return element.length>0 && !element.match(/\n? *$/);
        }

        public function nextToken():String{
            return tok = tokens.length==0 ? null : tokens.shift();
        }

        public function token():String {
            return tok;
        }

        public function skip(sym:String){
            if(token()==sym)
                return nextToken();
            else
                throw new ArgumentError("skip:"+sym+" expected but "+token()+ " found");
        }
    }
}

1 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.
2 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.
3 Defines a function to match empty strings and those containing only spaces.
4 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.
5 Returns the current token.
6 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 10.28. [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 {
        var doc:XML;

        5     function DOMExpand(s:String){
            var st:CompactTokenizer = new CompactTokenizer(s);
            var rootName:String;
            while (st.nextToken()!="[") rootName=st.token();
            doc = expand(st,rootName);

        }

        public function toString():String {
            return doc.toXMLString();
        }

        15    function expand(st:CompactTokenizer,elementName:String):XML{
            var elem:XML= <{elementName}/>
            st.skip("[");
            while(st.token()=="@"){
                var attName:String=st.skip("@");
                st.nextToken();
                elem.@[attName]=st.skip("[");
                st.nextToken();
                st.skip("]");
            }
            while (st.token()!=""]"){
                var s:String = st.token().replace(/^\s*(.*?)\s*$/,"$1");
                st.nextToken();
                if(st.token()=="[")
                    elem.appendChild(expand(st,s));
                else
                    elem.appendChild(s);
            }
            st.skip("]");
            return elem;
        }
    }
}
```

1 Defines a class with a single instance variable: the XML document to create.
2 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.
3 Creates a new XML structure by calling `expand` on the name of the root element.
4 The serialization of the XML structure is the string representation of the `doc` instance variable.
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.

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.

Loops on each child. Gets the string that follows, which is then trimmed using a regular expression.

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.

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.
10.5. XML processing with Swift

Swift [83] 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.

```swift
import Foundation

func processDOM(_ string:String)->String{
    var res=""
    func compact(_ nodeIn:XMLNode?,_ indent:String){
        guard (nodeIn != nil) else {return}
        let node = nodeIn!
        var newIndent=indent
        switch node.kind {
        case .element:
            let name = node.name!
            let elem = node as! XMLElement
            res += name+
            newIndent += String(repeating:" ", count: name.count+1)
            var first = true
            if elem.attributes != nil {
                for attr in elem.attributes! {
                    if !first {res += "\n\n(newIndent)}
                    res += attr.name!\(attr.stringValue!)]\(attr.stringValue!)]"
                    first = false
                }
            }
            if elem.children != nil {
                for child in elem.children! {
                    if !first {res += "\n\n(newIndent)"
                    compact(child, newIndent)
                    first = false
                }
            }
            res += "]"
        case .text:
```
Imports the Foundation framework needed for XML processing.

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.

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.

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.

Loops over the list of children possibly changing line if it not the first child.

Recursively calls compact on the child node with the updated indentation.

Processes a text node by adding it to res after removing starting and endind spaces, tabs and newlines.

Parses the file by creating a new XML document from the content of the input string.

Calls the compacting process on the document node with an empty indentation string and ends the last line with a newline.

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

// Caution: internal entities are not dealt properly with this API.
func processSAX(string:String)->String {

5     class xmlParserDelegate: NSObject, XMLParserDelegate {
         var res=""
         var closed=false
         var indent=""

10         var lastNodeWasText=false

     func parser(_ parser: XMLParser,
             didStartElement elementName: String,
             namespaceURI: String?, qualifiedName qName: String?,
         attributes attributeDict: [String : String] = [:]){

15             if closed {
                 res += "\n\n\n"(indent)"
                 closed = false
             }

20             indent += String(repeating: " ", count:elementName.count+1)

         res += "\"\(elementName)\""
         var first = true
         for (name,val) in attributeDict {

25                 if !first {res += "\n\n\n"(indent)"}
         res += "@\(name)\"\(val)"
         first=false
         closed=true

30         }

     lastNodeWasText=false

     }

35     func parser(_ parser: XMLParser,
             didEndElement elementName: String,
             namespaceURI: String?, qualifiedName qName: String?){

         res += "\"
         closed=true
         indent=indent.dropLast(elementName.count+1)
     lastNodeWasText=false

30 }

40     func parser(_ parser: XMLParser,
             foundCharacters string: String){

        let s=string.trimmingCharacters(in: .whitespacesAndNewlines)
        if s.count>0 { (Example 2.2)
```
if closed && !lastNodeWasText{
    res += "\n\n(indent)"
    closed = false
}
closed=true
res += "\(s)"
}
lastNodeWasText=true
}

public func parserDidEndDocument(_ parser: XMLParser){
    res += "\n"
}

func parser(_ parser: XMLParser,
    parseErrorOccurred parseError: Error){
    print("** parseErrorOccurred:(parseError)")
}

func parser(_ parser: XMLParser,
    validationErrorOccurred validationError: Error){
    print("** validationErrorOccurred:(validationError)")
}

let myDelegate=xmlParserDelegate()
let parser=XMLParser(data: string.data(using: .utf16)!
parser.delegate = myDelegate
parser.parse()
return myDelegate.res
}

func processSAX(path:String)->String{
    return processSAX(string:try! String(contentsOf: URL(fileURLWithPath: path)))
}

Imports the Foundation framework for the XML processing.
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.
When a new element is started, finishes the current indentation if needed.
Updates the current indentation by adding the length of the element name.
Prints the first attribute on the same line and the others on the subsequent lines properly indented.
When an element finishes, closes the current bracket and updates the current indentation.
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.
When the document parsing finishes, add a terminating newline.
Deals with a parsing error by printing an error message.
Creates an instance of the SAX parser delegate.
Creates an instance of the SAX parser.
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.

```swift
1 import Foundation

class CompactTokenizer {

    var tokens: [String]

    var iTok: Int

    init(string: String) {
        let pattern = "\[|\]|@|[^\[\]|@\n]+"
        let regex = try! NSRegularExpression(pattern: pattern, options: [])
        let nsrange = NSRange(string.startIndex..<string.endIndex, in: string)

        iTok = 0
        tokens = regex.matches(in: string, options: [], range: nsrange).map { match in
            let matchedRange = Range(match.range(at: 1), in: string)!
            return string[matchedRange].trimmingCharacters(in: .whitespacesAndNewlines)
        }.filter {!$0.isEmpty}
    }

    func nextToken() -> String {
        let token = tokens[iTok]
        iTok += 1
        return token
    }

    func skip(_ string: String) {
        if nextToken() != string {
            // skip
        }
    }
}
```

Alternative approaches

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/
print("(string) expected at position \(iTok-1\)")
)

func isAtEnd()->Bool {
    return iTok>=tokens.count
}

func expand(string:String) -> XMLNode {
  let ct=CompactTokenizer(string: string)
  func expandElem(_ elem:XMLElement,_ inTok:String){
    var tok = inTok
    while tok == "@" {
      let attName=ct.nextToken()
      ct.skip("[")
      elem.addAttribute(
        XMLNode.attribute(withName: attName,
          stringValue: ct.nextToken()) as! XMLNode)
      ct.skip("]")
      tok=ct.nextToken()
    }
    while tok != ""]" && !ct.isAtEnd() {
      let s=tok
      let nextTok=ct.nextToken()
      if  nextTok == "[" {
        let child = XMLNode.element(withName: s) as! XMLElement
        expandElem(child,ct.nextToken())
        elem.addChild(child)
        tok = ct.nextToken()
      } else {
        elem.addChild(XMLNode.text(withStringValue: s) as! XMLNode)
        tok = nextTok
      }
    }
    }
    }

    var rootName=""
    var token=ct.nextToken()
    while token != "][" {
      rootName = token
      token = ct.nextToken()
    }
    let root = XMLElement(name: rootName)
    let doc=XMLDocument(rootElement: root)
    expandElem(root,ct.nextToken())
    return doc
}

func expand(path:String)->XMLNode {
    return expand(string: try! String(contentsOf: URL(fileURLWithPath: path)))
}
Imports the Foundation framework for the XML processing.

Defines the CompactTokenizer class.

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.

Returns the next token.

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.

Checks if there are still tokens left to process.

Creates an XML document corresponding to the content of the input string.

Creates an instance of the CompactTokenizer class that will be used to get the tokens.

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.

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.

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.

A child node is expanded by a recursive call whose result is added as the last child of the current element.

A text node is added as the last child of the current element.

The name of the root is the first token immediately followed by an opening square bracket.

Create the document node by calling XMLDocument with an element that serves as its root.

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 pretty printed string using the xmlString(options: .nodePrettyPrint) method.
10.6. 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.

10.6.1. 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 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 (true or false) and null;

- **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 a not a legal Javascript identifier, the 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 10.32 and its JSON equivalent in Example 10.33. 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 10.33 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 10.32. [WineList.xml] A small wine list in XML

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

1 Root element of a small data-oriented XML document.
2 First wine element, with two attributes.
3 An attribute with a string value.
4 An element with a boolean value.
5 An element with a numerical value.
6 Second wine element with two attributes.

Example 10.33. [WineList.json] A small wine list in JSON

The same content as in Example 10.32, transformed in JSON, the same callout numbers in both listings correspond to the same elements.

```json
1 {
   "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
         },
         {
            "name": "Riesling Hugel",
            "appellation": "Alsace",
            "is-red": false,
            "origin": {
               "country": "France",
               "region": "Alsace and East"
            },
            "price": 17.95,
            "year": 2002
         }
      ]
   }
```

Alternative approaches

Visit us at http://www.renderx.com/
Root element translated as a single key-value pair.

All wines will be stored under a single key corresponding to an array of values.

An attribute is transformed into a key-value pair with a string value within quotes.

A boolean value is written without quotes.

A numerical value is written without quotes.

Second wine of the array.

The JSON version given above was produced by Example 10.34, 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 10.34. [xml2json.xsl] XSLT stylesheet to convert an XML file into JSON.

```xml
<!DOCTYPE stylesheet [ ]>
<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="*"/>
    <xsl:output omit-xml-declaration="yes"/>
    <xsl:template match="/">
    </xsl:template>
    <!-- end current line, indent $str by $nb spaces -->
    <xsl:template name="doIndent">
        <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>
</xsl:stylesheet>
```
Alternative approaches

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-->  
<xs:template name="outArray">  
<xs:param name="indent" as="xsd:integer"/>  
<xs:call-template name="doIndent">  
<xs:with-param name="nb" select="$indent+3" as="xsd:integer"/>  
<xs:with-param name="str" select="concat(gl:jsString(string(current-grouping-key())), ':[')"/>  
</xs:call-template>  
<!-- output its contents "recursively" -->  
<xs:apply-templates select="current-group()">  
<xs:with-param name="indent" select="$indent+6" as="xsd:integer"/>  
<xs:with-param name="outputName" select="false()"/>  
</xs:apply-templates>  
<xs:call-template name="doIndent">  
<xs:with-param name="nb" select="$indent+3" as="xsd:integer"/>  
<xs:with-param name="str" select="']'"/>  
</xs:call-template>  
</xs:template>  

55 <!-- an attribute is its name followed by its text value -->  
<xs:template match="@*">  
<xs:param name="indent" as="xsd:integer"/>  
<xs:call-template name="doIndent">  
<xs:with-param name="nb" select="$indent" as="xsd:integer"/>  
<xs:with-param name="str" select="concat(gl:jsString(name()), ': ' , gl:jsString(.))"/>  
</xs:call-template>  
</xs:template>  

65 <!-- a text node is the value of its content -->  
<xs:template match="text()">  
<xs:value-of select="gl:jsString(normalize-space(.))"/>  
</xs:template>  

70 <!-- any child node -->  
<xs:template match="*">  
<xs:param name="indent" as="xsd:integer"/>  
<xs:param name="outputName" select="true()"/>  
<!-- output the name of the tag if so desired -->  
<xs:if test="$outputName">  
<xs:call-template name="doIndent">  
<xs:with-param name="nb" select="$indent" as="xsd:integer"/>  
<xs:with-param name="str" select="gl:jsString(name())"/>  
</xs:call-template>  
</xs:if>  
</xs:template>
<xsl:choose>
  <xsl:when test="count(current-group())=1">
    <xsl:apply-templates select="."/>
    <xsl:with-param name="indent" select="$indent+3"/>
  </xsl:when>
  <xsl:otherwise>
    <!-- many element groups within braces-->
    <xsl:call-template name="outArray">
      <xsl:with-param name="indent" select="$indent"/>
    </xsl:call-template>
  </xsl:otherwise>
</xsl:choose>
<xsl:if test="position() != last()">
  <xsl:text>,</xsl:text>
</xsl:if>
</xsl:for-each-group>
</xsl:otherwise>
</xsl:choose>
<xsl:if test="position() = last()">
  <xsl:text>,</xsl:text>
</xsl:if>
</xsl:template>
</xsl:stylesheet>

1 Defines an entity for ending a line with a carriage return.
2 Starts the transformation of all node with an indentation of 0.
Example 10.34 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 10.34 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 10.34 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.
10.6.2. YAML

Another data structure notation that has been proposed is YAML[88], 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 10.35 shows how the content of our small wine list (Example 10.32) 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 10.35. [WineList.yaml] YAML version of Example 10.32

In order to show the parallel between XML, JSON and YAML, we use the same callout numbers as for Example 10.33.

```yaml
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
  - name: Riesling Hugel
    appellation: Alsace
    is-red: false
    origin:
      region: Alsace and East
      country: France
    price: 17.95
    year: 2002
```

1 Root element on a single line with no indentation.
2 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.
Alternative approaches

- A key with a string value. A string is not put within quotes unless it includes some special characters like quotes or newlines.
- A key with a boolean value.
- A key with a numerical value.
- 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.
10.7. Additional information on alternative approaches


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 [74] and the XML modules are described in a section of the online manual [75].

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.

Swift [83] 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 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 as 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


Bibliography


**Web Sites**


[64] *libxml++*, http://libxmlplusplus.sourceforge.net/

[65] *eXist-db Open Source Native XML Database*, http://exist-db.org/

Bibliography


[73] OWL 2 Web Ontology Language, http://www.w3.org/TR/owl-overview/


[76] Resource Description Framework (RDF), http://www.w3.org/RDF/

[77] RDF Validation service, http://www.w3.org/RDF/Validator/


[81] XSLT 2.0 and XQuery processors, http://www.saxonica.com/documentation/about/intro.xml


[83] Swift programming language, https://swift.org


[88] YAML Ain't a Markup Language, http://www.yaml.org/
## Appendix A. Some XML Related Technologies and Systems

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full name</th>
<th>Sections</th>
<th>Specs</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOM</td>
<td>Document Object Model</td>
<td>Section 8.1</td>
<td>[47]</td>
</tr>
<tr>
<td>DTD</td>
<td>Document Type Definition</td>
<td>Section 3.1</td>
<td>[12]</td>
</tr>
<tr>
<td>E4X</td>
<td>EcmaScript for XML</td>
<td>Section 10.4</td>
<td>[66]</td>
</tr>
<tr>
<td>JSON</td>
<td>Javascript Object Notation</td>
<td>Section 10.6.1</td>
<td>[69]</td>
</tr>
<tr>
<td>PDF</td>
<td>Portable Document Format</td>
<td>Section 5.4</td>
<td>[1]</td>
</tr>
<tr>
<td>RELAX NG</td>
<td>REgular LAnguage for XML, New Generation</td>
<td>Section 3.3</td>
<td>[15]</td>
</tr>
<tr>
<td>SAX</td>
<td>Simple Application programming interface for Xml</td>
<td>Section 8.2</td>
<td>[38]</td>
</tr>
<tr>
<td>SGML</td>
<td>Standard Generalized Markup Language</td>
<td>Chapter 1</td>
<td>[26]</td>
</tr>
<tr>
<td>URI</td>
<td>Uniform Resource Identifier</td>
<td>Section 2.1</td>
<td>[48]</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locator</td>
<td>Section 2.1</td>
<td>[49]</td>
</tr>
<tr>
<td>XML</td>
<td>eXtended Markup Language</td>
<td>*</td>
<td>[12]</td>
</tr>
<tr>
<td>XML Schema</td>
<td>XML Schema</td>
<td>Section 3.2</td>
<td>[45],[9]</td>
</tr>
<tr>
<td>XPath</td>
<td>XML PATH language</td>
<td>Chapter 4</td>
<td>[17]</td>
</tr>
<tr>
<td>XSL</td>
<td>eXtensible Stylesheet Language</td>
<td>Section 5.1</td>
<td>[7]</td>
</tr>
<tr>
<td>XSLT</td>
<td>XSL Transformations</td>
<td>Section 5.1</td>
<td>[28]</td>
</tr>
<tr>
<td>YAML</td>
<td>YAML Ai’nt a Markup Language</td>
<td>Section 10.6.2</td>
<td>[88]</td>
</tr>
</tbody>
</table>

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at [http://www.renderx.com/](http://www.renderx.com/)
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

```
<!DOCTYPE rootElement SYSTEM "file name" {[ !ENTITY *]}? «>»
<!ELEMENT NCName «(» (#PCDATA «|»)? regexpOf element name «)» «>»
<!ELEMENT NCName (#PCDATA) «>»
<!ELEMENT NCName EMPTY «>»
<!ATTLIST elementNCName attributeNCName declValue default «>»
          declValue = CDATA | ID | IDREF | «(» CNAME+ «)»
          default = {#REQUIRED | #IMPLIED}
<![CDATA[ ... ]]«>»
<!ENTITY name "" ... "" «>»
<!ENTITY % name "" ... "" «>»
<!ENTITY name SYSTEM ""file name"" «>»
```
B.3. XML Schema

Table B.3.  Table 3.2, Section 3.2

```xml
<xs:schema targetNamespace="URI">
   xs:import* (xs:simpleType | xs:complexType | xs:element | xs:group)*
</xs:schema>

<xs:import nameSpace="URI"
   schemaLocation="URI"/>

<xs:simpleType name="NCName">
   xs:restriction
</xs:simpleType>

<xs:complexType name="NCName">
   mixed="true">
      {xs:choice | xs:sequence | xs:group}? xs:attribute*
</xs:complexType>

<xs:element name="QName" type="TName"/>

<xs:element name="QName" ref="EName"/>

<xs:element name="QName">
   {xs:simpleType | xs:complexType}?
   {xs:unique | xs:key | xs:keyref}*
</xs:element>

<xs:sequence {min|max}occurs="nonNegativeInteger|unbounded">
   {xs:element | xs:choice | xs:sequence | xs:group}*
</xs:sequence>

<xs:choice {min|max}occurs="nonNegativeInteger|unbounded">
   {xs:element | xs:choice | xs:sequence | xs:group}*
</xs:choice>

<xs:group name="NCName">
   {xs:choice | xs:sequence}*
</xs:group>

<xs:attribute name="NCName" type="TName" use="required"/>

<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>

<xs:{unique|key} name="NCName">
   xs:selector xs:field+
</xs:{unique|key}>

<xs:keyref name="NCName"
   refer="NCName">
   xs:selector xs:field+
</xs:keyref>

<xs:{selector|field} xpath="XPathExpr"/>
```
### B.4. RELAX NG

#### Table B.4.  Table 3.3, Section 3.3

<table>
<thead>
<tr>
<th>Compact Syntax (RNC)</th>
<th>XML Syntax (RNG)</th>
</tr>
</thead>
<tbody>
<tr>
<td>`{default? namespace id=URI</td>
<td><code>&lt;grammar&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;start&gt; pattern &lt;/start&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;define name=&quot;NCName&quot;&gt;pattern+&lt;/define&gt;)*</code></td>
</tr>
<tr>
<td>`{ start=pattern</td>
<td><code>&lt;grammar&gt;</code></td>
</tr>
</tbody>
</table>
| | `}`*
| `| id=pattern `* |

**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`:
  `<data {type="NCName">`|
  `{<param name="NCName">string</param>}*` |
  `}</data>`
## B.5. Schematron

### Table B.5. Table 3.4, Section 3.4

```
<schema targetNameSpace="URI">
    title? ns? pattern+
</schema>

<ns prefix="QName">
    uri="URI"/>
</ns>
<title>
    PCDATA
</title>
<pattern abstract="yes | no">
    id="ID"
    is-a="IDREF">
        param* rule+
    </pattern>

<param name="QName">
    value="XPathExpr"/>
</param>
<brule context="XPathExpr">
    {let | report | assert}*
</brule>

<let name="QName">
    value="XPathExpr"/>
</let>
<report test="XPathExpr">
    {PCDATA | value-of | name}* 
</report>
<assert test="XPathExpr">
    {PCDATA | value-of | name}* 
</assert>
<value-of select="XPathExpr"/>
<name/>
```
B.6. XSLT

Table B.6. Table 5.1, Section 5.1

```
<xsl:stylesheet>
  xsl:import*, (declaration|xsl:variable|xsl:param)*
</xsl:stylesheet>

<xsl:template match="pattern" name="QName">
  xsl:param*, sequence-constructor*
</xsl:template>

<xsl:param name="QName" select="expression">
  sequence-constructor
</xsl:param>

<xsl:apply-templates select="expression">
  (xsl:sort*|xsl:with-param)*
</xsl:apply-templates>

<xsl:call-template name="Qname"/>

<xsl:with-param name="QName" select="expression">
  sequence-constructor
</xsl:with-param>

<xsl:function name="QName">
  xsl:param*, sequence-constructor*
</xsl:function>

<xsl:value-of select="expression">
  sequence-constructor
</xsl:value-of>

<xsl:variable name="QName" select="expression">
  sequence-constructor
</xsl:variable>

<xsl:copy>
  sequence-constructor
</xsl:copy>

<xsl:copy-of select="expression"/>

<xsl:if test="expression">
  sequence-constructor
</xsl:if>

<choose>
  xsl:when*, xsl:otherwise?
</choose>

<xsl:when test="expression">
  sequence-constructor
</xsl:when>

<xsl:otherwise>
  sequence-constructor
</xsl:otherwise>

<xsl:for-each select="expression">
  xsl:sort*, sequence-constructor
</xsl:for-each>

<xsl:for-each-group select="expression" group-by="expression">
  xsl:sort*, sequence-constructor
</xsl:for-each-group>
```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;xsl:sort select=&quot;expression&quot; data-type=&quot;{string}&quot;&gt;</code></td>
<td><code>&lt;xsl:sort&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;xsl:key name=&quot;qname&quot; match=&quot;pattern&quot; use=&quot;expression&quot;&gt;</code></td>
<td><code>&lt;xsl:key name=&quot;qname&quot; match=&quot;pattern&quot; use=&quot;expression&quot;&gt;</code></td>
<td><code>&lt;xsl:sort&gt;</code></td>
</tr>
<tr>
<td><code>&lt;xsl:element name=&quot;{string}&quot;&gt;</code></td>
<td><code>&lt;xsl:element name=&quot;{string}&quot;&gt;</code></td>
<td><code>&lt;xsl:sort&gt;</code></td>
</tr>
<tr>
<td><code>&lt;xsl:text&gt;</code></td>
<td><code>&lt;xsl:text&gt;</code></td>
<td><code>&lt;xsl:sort&gt;</code></td>
</tr>
<tr>
<td><code>&lt;xsl:attribute name=&quot;{string}&quot; select=&quot;expression&quot;&gt;</code></td>
<td><code>&lt;xsl:attribute name=&quot;{string}&quot; select=&quot;expression&quot;&gt;</code></td>
<td><code>&lt;xsl:sort&gt;</code></td>
</tr>
<tr>
<td><code>&lt;xsl:attribute-set name=&quot;QName&quot; use-attribute-sets=&quot;Qnames&quot;&gt;</code></td>
<td><code>&lt;xsl:attribute-set name=&quot;QName&quot; use-attribute-sets=&quot;Qnames&quot;&gt;</code></td>
<td><code>&lt;xsl:sort&gt;</code></td>
</tr>
<tr>
<td><code>&lt;xsl:message&gt;</code></td>
<td><code>&lt;xsl:message&gt;</code></td>
<td><code>&lt;xsl:sort&gt;</code></td>
</tr>
</tbody>
</table>
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 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 \(\texttt{C.1}. \) [\texttt{DBTemplates.xml}] Examples of the most common uses of DocBook 5 elements in this document.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- example uses of the most common docbook templates -->
<?oxygen RNGSchema="DBx.rnc" type="compact"?>
<!DOCTYPE book [ 
<!ENTITY % ralient SYSTEM "rali.ent">
 %ralient; ]>
<chapter
 xmlns="http://docbook.org/ns/docbook"
 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>
<info>
 <keywordset><keyword>keyword for this document</keyword></keywordset>
</info>
<indexterm class="startofrange" xml:id="section_start">
 <primary>instance document</primary>
</indexterm>
<!-- range of the section -->
<indexterm class="endofrange" startref="section_start"/>
<section xml:id="sec_title">
 <title>Title</title>
 <indexterm><primary>index term</primary></indexterm>
</section>
</chapter>
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/
<!-- figure -->
<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 appear in HTML -->
      <imagedata fileref="images/WineTree.png" contentwidth="8in" width="8in"/>
    </imageobject>
    <imageobject role="fo">
      <imagedata fileref="images/WineTree.pdf" scale="75"/>
    </imageobject>
  </mediaobject>
  <caption><simpara/></caption><!-- if a line is wanted at the bottom -->
</figure>

<!-- table -->
<table xml:id="tab_NAME" shortentry="1">
  <title>Title</title>
  <titleabbrev>Title small</titleabbrev>
  <!-- to force a table to be on the same page -->
  <tgroup cols="2" align="left">
    <tbody>
      <row>
        <entry>Row1_Col1</entry>
        <entry>Row1_Col2</entry>
      </row>
      <row>
        <entry>Row2_Col1</entry>
        <entry>Row2_Col2</entry>
      </row>
    </tbody>
  </tgroup>
  <caption><simpara>Caption</simpara></caption>
</table>

<!-- example listing -->
<example xml:id="fileName.ext">
  <title>[[link xl:href="&r-root;fileName.ext"><filename>fileName.ext</filename></link> a title</title>
  <indexterm>
    <primary>Examples</primary>
    <secondary>fileName.ext</secondary>
  </indexterm>
  <simpara>Caption to appear at the example.</simpara>
</example>

<!-- listing with callouts -->
<programlistingco>
  <areaspec units="linecolumn">
    <area xml:id="co_id" coords="L 70"/>
  </areaspec>
  <programlisting linenumbering="numbered">
    <!----<xi:include href="file..." parse="text"/>--></programlisting>
  <calloutlist>
    <callout arearefs="co_id">
      <para>XXX</para>
    </callout>
  </calloutlist>
</programlistingco>
</callout>
</calloutlist>

105   </programlistingco>
</example>

<para> <!-- things that can appear in a paragraph... -->
   <!-- references to line numbers in a listing (personal extension to docbook -->
   <!--<refline lineid="co_id"/> -->
   <!--<refline lineid="co_id" source="yes"/>-->  
   <refline lineid="co_id" linkend="fileName.ext"/>
   <!-- italics and bold face -->
   <emphasis>italics</emphasis><emphasis role="strong">bold</emphasis>
   <!-- link to an external url -->
   <link xl:href="http:..."/>
   <link xl:href="http:...">TEXT</link>
   <!-- references to an ID, 
   can also be used for refering to a bibliography entries -->
   <xref linkend="ID"/>
   <xref linkend="ID" xrefstyle="select: label page"/>
   <!-- index terms -->
   <indexterm class="startofrange" xml:id="PRIMARY_start">  
   <primary>PRIMARY</primary>
   </indexterm>
   <indexterm class="endofrange" startref="PRIMARY_start"/>
   <indexterm><primary>PRIMARY</primary></indexterm>
</para>
</section>

<!-- forcing a page break or clearing floats--> 
<?hard-pagebreak?>
<?float-clear?>

140

<!-- bibliography entry -->
<bibliography>
   <biblioentry xml:id="ID">
      <authorgroup>
         <author>
            <personname>
               <firstname>FIRSTNAME</firstname>
               <surname>NAME</surname>
            </personname>
         </author>
      </authorgroup>
      <publishername>PUB</publishername>
      <pubdate>YYY</pubdate>
      <title>TITRE</title>
   </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

```xml
1 <cellar-book noNamespaceSchemaLocation="CellarBook.xsd">
2   <wine-catalog>
3     <wine name="Domaine de l'Île Margaux"
4       appellation="Bordeaux supérieur"
5       classification="a.c."
6       code="C00043125"
7       format="750ml">
8       <properties>
9         <color>red</color>
10        <alcoholic-strength>12.5</alcoholic-strength>
11        <nature>still</nature>
12     </properties>
13     <origin>
14       <country>France</country>
15       <region>Bordeaux</region>
16       <producer> SCEA Domaine de L'Île Margaux (B.P. 5) </producer>
17     </origin>
18     <comment>Ready for drinking now</comment>
19     <food-pairing> Accompanies <emph>Bordelaise ribsteak</emph>, <bold>pork with prunes</bold> or magret de canard. </food-pairing>
20     <price>22.80</price>
21     <year>2002</year>
22   </wine>
23   <wine name="Riesling Hugel"
24     appellation="Alsace"
25     classification="a.c."
26     code="C00042101"
27     format="750ml">
28     <properties>
29         <color>white</color>
30         <alcoholic-strength>12</alcoholic-strength>
31         <nature>still</nature>
32     </properties>
33     <origin>
34       <country>France</country>
35       <region>Alsace and East</region>
36       <producer> Hugel & Fils</producer>
37     </origin>
38     <price>17.95</price>
39     <year>2002</year>
40   </wine>
41   <wine name="Château Montaguëret"
42     appellation="Anjou"
43     classification="a.c."
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at [http://www.renderx.com/](http://www.renderx.com/)
<wine name="Château de Montgue" code="C00871996" format="750ml">
  <properties>
    <color>rosé</color>
    <alcoholic-strength>11</alcoholic-strength>
    <nature>still</nature>
  </properties>
  <origin>
    <country>France</country>
    <region>Loire Valley</region>
    <producer>SCEA Château de Montguéret</producer>
  </origin>
  <comment> Made with Grolleau (100%). Ready to drink now. Serve at 8º-10ºC. </comment>
  <tasting-note>Tender pink in color, this wine shows light raspberry highlights. </tasting-note>
  <price>14.65</price>
  <year>2003</year>
</wine>

<wine name="Mumm Cordon Rouge" appellation="Champagne" code="C00312363" format="375ml">
  <properties>
    <color>white</color>
    <alcoholic-strength>12</alcoholic-strength>
    <nature>Champagne</nature>
  </properties>
  <origin>
    <country>France</country>
    <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 delicate and has exquisite bubbles. </tasting-note>
  <price>33.00</price>
  <year>2000</year>
</wine>

<wine name="Prado Rey Roble" appellation="Ribera-del-duero" classification="d.o." code="C00929026" format="magnum">
  <properties>
    <color>red</color>
    <alcoholic-strength>12.5</alcoholic-strength>
    <nature>still</nature>
  </properties>
  <origin>
    <country>Spain</country>
    <region>Old Castille</region>
    <producer>Real Sitio de Ventosilla SA</producer>
  </origin>
</wine>
<price>35.25</price>

100 <year>2002</year>
</wine>
</wine-catalog>
<owner>

105 <name>
  <first>Jude</first>
  <family>Raisin</family>
</name>

110 <street>1234 rue des Châteaux</street>
<city>St-George</city>

115 <province>ON</province>
<postal-code>M7W 7S0</postal-code>
</owner>
<location>
  <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>&lt;cat:bold&gt;Guy Lapalme, Montréal&lt;/cat:bold&gt;: should reorder soon</comment>
</wine>

125 <wine code="C00312363">
  <purchaseDate>2004-11-19</purchaseDate>
  <quantity>5</quantity>
  <rating>3</rating>
  <comment>Bottle too small...</comment>
</wine>

130 <wine code="C00871996">
  <purchaseDate>2005-06-19</purchaseDate>
  <quantity>0</quantity>
  <comment>Really great</comment>
</wine>

135 <wine code="C00929026">
  <purchaseDate>2003-10-15</purchaseDate>
  <quantity>1</quantity>
  <comment>for &lt;cat:bold&gt;big&lt;/cat:bold&gt; parties</comment>
</wine>

140 </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.xsd">
   <wine name="Domaine de l'Île Margaux"
      appellation="Bordeaux supérieur"
      classification="a.c."
      code="C00043125"
      format="750ml">
      <properties>
         <color>red</color>
         <alcoholic-strength>12.5</alcoholic-strength>
         <nature>still</nature>
      </properties>
      <origin>
         <country>France</country>
         <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>, <bold>pork with prunes</bold> or magret de canard.
      </food-pairing>
      <price>22.80</price>
      <year>2002</year>
   </wine>
   <wine name="Riesling Hugel"
      appellation="Alsace"
      classification="a.c."
      code="C00042101"
      format="750ml">
      <properties>
         <color>white</color>
         <alcoholic-strength>12</alcoholic-strength>
         <nature>still</nature>
      </properties>
      <origin>
         <country>France</country>
         <region>Alsace and East</region>
         <producer>Hugel & Fils</producer>
      </origin>
      <price>17.95</price>
      <year>2002</year>
   </wine>
   <wine name="Château Montguéret"
      appellation="Anjou"
      classification="a.c."
      code="C00871996"
      format="750ml">
      <properties>
      </properties>
      <origin>
         <country>France</country>
         <region>Alsace and East</region>
         <producer>Hugel & Fils</producer>
      </origin>
      <price>17.95</price>
      <year>2002</year>
   </wine>
</wine-catalog>
<color>rosé</color>
<alcoholic-strength>11</alcoholic-strength>
<nature>still</nature>
</properties>
<origin>
  <country>France</country>
  <region>Loire Valley</region>
  <producer>SCEA Château de Montgéret</producer>
</origin>
<comment>
  Made with Grolleau (100%). Ready to drink now.
  Serve at 8º-10ºC.
</comment>
<tasting-note>
  Tender pink in color, this wine shows
  <emph>light raspberry</emph> highlights.
</tasting-note>
<price>14.65</price>
<year>2003</year>
</wine>
<wine name="Mumm Cordon Rouge"
      appellation="Champagne"
      classification="a.c."
      code="C00312363"
      format="375ml">
  <properties>
    <color>white</color>
    <alcoholic-strength>12</alcoholic-strength>
    <nature>Champagne</nature>
  </properties>
  <origin>
    <country>France</country>
    <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 delicate
    and has exquisite bubbles.
  </tasting-note>
  <price>33.00</price>
  <year>2000</year>
</wine>
<wine name="Prado Rey Roble"
      appellation="Ribera-del-duero"
      classification="d.o."
      code="C00929026"
      format="magnum">
  <properties>
    <color>red</color>
    <alcoholic-strength>12.5</alcoholic-strength>
    <nature>still</nature>
  </properties>
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, 157, 207, 217
algorithmic checking, 43
alternative notations for XML, 207
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

c 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
cocurrence checking, 43
CSS, 93-94

d D
DocBook, xvii, 231
DOCTYPE, 11
document creation, 145
document nodes, XPath, 51
document Object Model, 127-128
document order, 2
document parsing
E4X, 195
PHP, 187
Python, 173
document parsing, DOM
PHP, 178
Python, 165
Ruby, 158
Swift, 200
document parsing, SAX
PHP, 180
Python, 166
Ruby, 159
Swift, 201
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 E
E4X, 157
document creation, 196-199
document parsing, 195-196
document processing, 193-199
type, 52
elementFormDefault, 35
elements, 2
type, 52
element, 11, 18
definition, 48
parameter, 19
predefined, 18
system, 19
ENTITY, 18
type, 52
elements
CBWC-RDF-S.rdf, 118
CBWC-RDF-S.rq, 125
Index

CBWC-RDF-S.ttl, 121
Cellarbook.dtd, 19
CellarBook.rnc, 39
CellarBook.sch, 45
Cellarbook.xml, 10, 237
  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, 160
compactHTML.php, 179
compactHTML.xq, 108
compactHTML.xsl, 78
CompactReader.java, 150
CompactTokenizer.as, 196
CompactTokenizer.java, 146
CompactTokenizer.php, 184
CompactTokenizer.py, 171
CompactTokenizer.rb, 161
DBTemplates.xml, 233
DOMCompact.as, 195
DOMCompact.java, 128
DOMCompact.php, 178
DOMCompact.py, 165
DOMCompact.rb, 158
DOMCompact.swift, 200
DOMExpand.as, 198
DOMExpand.java, 147
DOMExpand.php, 186
DOMExpand.py, 172
DOMExpand.swift, 204
ETCompact.py, 174
ETExpand.py, 175
expand.rb, 163
JTreeHandler.java, 142
NamespaceExample.xml, 13
SAXCompact.java, 133
SAXCompact.php, 181
SAXCompact.py, 166
SAXCompact.rb, 159
SAXCompact.swift, 202
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, 240
WineCatalog.xq, 101
WineCatalog.xsd, 29
WineCatalog.xsl, 66
WineList.json, 208
WineList.xml, 208
WineList.yaml, 213
xml2json.xsl, 209
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, 231
  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

244
createElement(.), 145
createTextNode(.), 145
DocumentImpl(), 145
setAttribute(.), 145
Javascript, 157
Javascript Object Notation (JSON), 157
JAXP, 127
JSON, 207-212
conversion from XML, 209
typing, 212
JSON (Javascript Object Notation), 157
JTree, 138

L
labeled tree, 7
LaTeX, 231
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, 217
<Xygen/>, 49

P
parameter entity, 19
PHP
document creation, 184-187
document parsing, 187-192
document processing, 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
calendar line, 112
RDF, 124
Turtle, 124
XML, 99

R
range, XPath, 53
RDF, 217
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, 158-164
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, 231
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, 204-206
document processing, 200-204
DOM document parsing, 200-201
SAX document parsing, 201-204

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, 212

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), 207
xi:include, 48
XML
  alternative notations, 207-214
  alternatives, 157
  conversion to JSON, 209
XML alternative
  JSON, 207
  YAML, 213
XML text editor, 17
XMLHTTPRequest, 207
xmlns, 15
XMLSpy, 49
XMLStreamWriter
  writeAttribute(.,.), 153
  writeCharacters(.), 153
  writeEndElement(), 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:otherwelse, 61
  xsl:sort, 61
  xsl:template, 60
  xsl:text, 62
  xsl:value-of, 61
  xsl:variable, 61
  xsl:when, 61
xsltproc, 96

Y

YAML, 213-214
YAML Ain't a Markup Language (YAML), 213
Yet Another Markup Language (YAML), 213